VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Automatizace testovacích sekvencí na bázi grafické prezentace a objektově orientovaného programování

Test sequence automation based on Graphical Presentation and Object-oriented Programming

Patrik Děcký

Bakalářská práce

Vedoucí práce: Ing. Pavel Kodytek

Ostrava, 2021

Abstrakt

Práce popisuje význam znovu-použitelnosti a modularity, které jsou klíčové pro dosažení dostatečné flexibility automatizovaného měřicího systému. Dále se práce zabývá návrhem a vývojem interaktivní rozšířitelné aplikace pro grafickou reprezentaci testovacích sekvencí automatického testeru. Aplikace je vytvořena s myšlenkou modularity, znovu-použitelnosti a rozšiřitelnosti. Návrh je podřízen cílové skupině uživatelů, a to lidem, kteří nemají praktickou zkušenost s programováním. Má umožnit rychleji a lépe pochopit skriptované testovací sekvence a zabránit vzniku syntaktických chyb při tvorbě těchto testovacích sekvencí s možností editace na dotykových obrazovkách v průmyslu.

Klíčová slova

testovací sekvence, změna požadavků, modularita, LabVIEW, objektově orientované programování, bakalářská práce

Abstract

This work describes the importance of reusability and modularity. These are the essential parts of achieving flexibility of automated measurement systems. This work aims to design and implement an interactive expandable application for the graphical representation of the test sequences. The application has been designed with the idea of modularity, reusability, and expandability. The target group is not programmers, but rather people with little experience in programming. It is intended to provide a faster and better understanding of scripted test sequences and prevent its users from making syntactic errors. Users should be able to edit test sequences on touch screens.

Key Words

test sequence, requirements change, modularity, LabVIEW, object-oriented programming, bachelor thesis

Rád bych tímto poděkoval Ing. Pavlu Kodytkovi za vedení této bakalářské práce. Za mnoho podnětných rad, které mě vedly při vypracování, jak teoretické, tak praktické části. Dík také patří za to, že pro mě vždy byl ochoten najít si čas ke konzultaci.

Obsah

Seznam použitých symbolů a zkratek 6

Seznam ilustrací 7

Seznam tabulek 9

Úvod 10

1. Role automatizovaných testerů v průmyslu, popis motivace pro tvorbu interaktivního vývojového prostředí pro vlastní skriptovací jazyky 11

1.1. Testování 11

1.2. Testovací sekvence, výrobní sekvence 11

1.3. Automatizace testování 11

1.4. Změny požadavků 12

1.5. Znovu-použitelnost 12

1.6. Automatický tester, Průmysl 4.0 13

1.7. Problémy s textově zobrazovanými testovacími sekvencemi 14

2. Popis problematiky objektově orientovaného programování, grafického vývojového prostředí a základní syntaxe jazyka použitého pro informační měřící systém 15

2.1. Objektově orientované programování 15

2.2. Grafické vývojové prostředí LabVIEW 20

2.3. Syntaxe skriptovacího jazyka 23

3. Koncepční návrh aplikace na základě posouzení vstupních a výstupních parametrů úlohy 30

3.1. Definice požadavků 30

3.2. Návrh aplikace 31

4. Vytvoření interaktivní aplikace 37

4.1. Struktura projektu 37

4.2. Grafický systém 39

4.3. Systém zobrazení příkazů 42

4.4. Grafické uživatelské rozhraní 43

4.5. Zpracování události 45

5. Ověření funkce a zhodnocení výsledků 47

5.1. Rozšiřitelnost, znovu-použitelnost 47

5.2. Práce se soubory 47

5.3. Editace souborů 47

5.4. Grafické zobrazení 47

Závěr 49
Literatura 50
Seznam příloh 51

Seznam použitých symbolů a zkratek

OOP — Objektově orientované programování
VI — Virtual Instrument (Virtuální přístroj)
HW — Hardware (Existující technické vybavení počítače)
SW — Software (Programové vybavení počítače)
GUI — Graphical User Interface (Grafické uživatelské rozhraní)
2D — Dvou dimenzionální
LV — LabVIEW

Seznam ilustrací

Obr. 1 — Modularita: návrh modulu lineárního posunu 13
Obr. 2 — OOP: hierarchie dědičnosti třídy auto 16
Obr. 3 — OOP: kompozice 17
Obr. 4 — OOP: Návrhový vzor – Strategy 19
Obr. 5 — OOP: Návrhový vzor – Observer 19
Obr. 6 — OOP: Abstract factory design pattern 20
Obr. 7 — LabVIEW: ukázka čelního panelu 21
Obr. 8 — LabVIEW: ukázka blokového diagramu 21
Obr. 9 — LabVIEW: porovnání s textovým zobrazením 22
Obr. 10 — LabVIEW: restrikce objektu stejné třídy 23
Obr. 11 — LabVIEW: řešení restrikce objektu stejné třídy 23
Obr. 12 — LabVIEW: restrikce objektu třídy s objektem této třídy 23
Obr. 13 — LabVIEW: řešení restrikce objektu třídy s objektem této třídy 23
Obr. 14 — Syntaxe jazyka: deklarace a definice proměnné 24
Obr. 15 — Syntaxe jazyka: spojená deklarace a definice 24
Obr. 16 — Syntaxe jazyka: matematické operace 24
Obr. 17 — Syntaxe jazyka: definice makra 25
Obr. 18 — Syntaxe jazyka: volání makra 25
Obr. 19 — Syntaxe jazyka: linkování maker 25
Obr. 20 — Syntaxe jazyka: linkování skriptu 26
Obr. 21 — Syntaxe jazyka: větvení 26
Obr. 22 — Syntaxe jazyka: operátor „rovná se“ 26
Obr. 23 — Syntaxe jazyka: skládání podmínek 27
Obr. 24 — Syntaxe jazyka: příkaz switch 27
Obr. 25 — Syntaxe jazyka: cyklus „do while“ 28
Obr. 26 — Syntaxe jazyka: cyklus „for“ 28
Obr. 27 — Syntaxe jazyka: skoky 28
Obr. 28 — Návrh: Use Case diagram 31
Obr. 29 — Návrh: struktura systému 32
Obr. 30 — Návrh: hierarchie modulu 33
Obr. 31 — Návrh: spouštění modulu 33
Obr. 32 — Návrh: rozložení GUI 34
Obr. 33 — Návrh: samostatný příkaz 35
Obr. 34 — Návrh: blok příkazů 36
Obr. 35 — Implementace: projekt 37
Obr. 36 — Implementace: inicializace systému 38
Obr. 37 — Implementace: asynchronní spuštění modulu 38
Obr. 38 — Implementace: hierarchie tisknutelných dat 40
Obr. 39 — Implementace: hierarchie tisknutelných objektů 40
Obr. 40 — Implementace: hierarchie tisknutelných oken 41
Obr. 41 — Implementace: vnoření oken 42

Obr. 42 — Implementace: GUI 43
Obr. 43 — Implementace: příkaz 44
Obr. 44 — Implementace: blok příkazů 44
Obr. 45 — Implementace: záložky 44
Obr. 46 — Implementace: zpracování události 46
Obr. 47 — Zhodnocení: porovnání textového a grafického zobrazení 48

Seznam tabulek

Tab. 1 — Syntaxe jazyka: datové typy 24
Tab. 2 — Syntaxe jazyka: matematické operace 24
Tab. 3 — Syntaxe jazyka: operátory porovnání 26
Tab. 4 — Syntaxe jazyka: logické operátory 27

Úvod

Trendem v automatizaci jsou univerzální konfigurovatelná zařízení oproti zařízením jednoúčelovým. Následkem je přibývání konfiguračních souborů různých typů a potenciálně vývoj vlastních skriptovacích jazyků. Díky skriptování je možné měnit chování automatizovaného testeru pro různé aplikace bez nutnosti měnit kód. Změna chování se provádí pouze změnou konfigurace.

Práce se zabývá popisem znovu-použitelnosti a modularity pro návrh univerzálních automatizovaných měřicích a výrobních systémů. Spolu s objektově orientovaným programováním to jsou klíčové koncepty pro návrh velkých flexibilních systémů. Z toho důvodu bude v práci popsán koncept objektově orientovaného programování, návrhové vzory a jejich podpora v LabVIEW.

Pro návrh aplikace je klíčové specifikování funkčních a nefunkčních požadavků a jejich následná analýza. Mezi hlavní požadavky na aplikaci patří integrovatelnost do automatizovaného testeru ATEsteru, grafické zobrazení příkazů, možnost otevření více skriptů, a hlavně rozšiřitelnost na další druhy textových souborů. K rozšiřitelnosti také patří schopnost reagovat na změnu syntaxe skriptovacího jazyka automatizovaného testeru. Cílem práce je kromě návrhu také implementace interaktivní aplikace v LabVIEW na základě provedeného návrhu. Při implementaci je nutno přihlédnout k faktu, že aplikace bude používaná v průmyslu na dotykových obrazovkách, a proto by grafické prvky a práce s nimi měly být přizpůsobeny pro ovládání bez použití hardwarové klávesnice a myši.

1. Role automatizovaných testerů v průmyslu, popis motivace pro tvorbu interaktivního vývojového prostředí pro vlastní skriptovací jazyky

Testování je nezbytnou součástí průmyslu, protože bez něj neexistuje žádný způsob, jak definovat kvalitu produktů. Bez automatizace může testování být velmi nákladné. Mělo by probíhat skrze celý proces výroby, protože čím dříve se vady objeví, tím více nákladů se ušetří na opravách. Tester by měl být navrhován modulárně a flexibilně tak, aby bylo snadné jej upravit při změně parametrů nebo požadavků a aby bylo případně možné co nejvíce části znovu použít. Optimální je navrhnutí vlastního skriptovacího jazyka, pomocí jehož se bude testovací sekvence řídit. Neexistují vývojová prostředí pro na míru vytvořené jazyky, a proto je dobré navrhnout a vytvořit vlastní editor na tvorbu skriptů takovýchto skriptovacích jazyků.

1.1. Testování

Testování je proces ověřování platnosti nějaké vlastnosti nebo schopnosti, ať už se jedná o písemné testování studentů ve škole nebo měření přesnosti ložiskové kuličky. Testování poskytuje informaci o kvalitě produktu. Pro mnoho aplikací je kritické, aby výrobky dosahovaly požadované jakosti. Každý výrobce by měl zaručovat určitou kvalitu a přesnost svých výrobků a tohoto dosáhne právě testováním.

Testování není jen ověřování, že systém funguje, ale je to také zjišťování chyb v systému, které musí být odstraněny. Produkty obsahující chyby mohou způsobit škody na majetku, nebo zranit lidi. Identifikování chyb dříve v procesu znamená výrazné snížení náročnosti a ceny oprav oproti odhalování chyb později[1]. Mnoho vad zjištěných při testování je možné opravit. Například pokud testování objeví, že tloušťka laku na výrobku je příliš malá, je možné výrobek nalakovat další vrstvou a tím tloušťku laku zvýšit. V takovém případě by bylo neekonomické celý výrobek vyhodit.

1.2. Testovací sekvence, výrobní sekvence

Otestování výrobku neznamená vyhodnocení pouze jednoho parametru. Častým požadavkem je otestování celé řady parametrů. Testovací sekvence je proces vykonávání jednotlivých testů včetně rozhodovacích operací.

Z pohledu programu je výrobní sekvence velice podobná jako sekvence testovací. Proces obsahuje jednotlivé kroky, které jsou vykonávány a mezi nimi dochází k rozhodovacím operacím. V průmyslu se často setkáváme s kombinací těchto dvou typů sekvencí. Již při výrobní sekvenci je nutné některé parametry otestovat, a z toho důvodu výrobní sekvence provede otestování daných parametrů. Není tedy nutné pohlížet na výrobní a testovací sekvence odlišně, protože se z hlediska výrobní/testovací aplikace jedná o prakticky stejné procesy.

Testovací sekvence je posloupnost kroků, která má za vstup neotestovaný produkt. Výstup je otestovaný produkt, informace o průběhu testů a vyhodnocení výsledků. Kroky v této posloupnosti mohou být samotné testy, manipulace s produkty, práce s daty a další.

1.3. Automatizace testování

Automatizace je využití logických programovacích příkazů a mechanizovaného vybavení k nahrazení rozhodovacího procesu a manuálních reakcí lidí na výzvy. Historicky mechanizace, jako použití

časovacích mechanismů k pohnutí páky, pomáhala lidem ve vykonávání fyzických úkolů. Automatizace posunuje mechanizaci ještě o jeden krok dále, značně redukuje potřebu lidských reakčních systémů a mentálních schopností a zároveň optimalizuje produktivitu.[3]

Automatizací se redukuje možnost vzniku lidské chyby a zvyšuje se tím spolehlivost testování, nebo výroby. Bez automatizace může být testování velmi nákladné. Počáteční náklady na automatizaci jsou vyšší, ale dlouhodobě se jedná o dobrou investici.

Automatické testování zahrnuje vykonávání před-skriptovaných testů pro vyhodnocení výsledků bez manuálního zásahu[2]. Pojem před-skriptovaný znamená, že sekvence je definovaná mimo zdrojový kód aplikace a je možné ji kdykoli změnit bez zásahu do zdrojového kódu. Takový automatický testovací systém provádí, upravuje, vyhodnocuje testovací sekvence i shromažďuje data bez nutné přítomnosti lidí. Pro uživatele automatického testovacího systému to znamená, že uživatel (ať už operátor výroby nebo výrobní linka) vloží neotestovaný výrobek a vyjmou otestovaný výrobek s informacemi o průběhu testování a celkovým výsledkem testu.

1.4. Změny požadavků

Vhodně navržený testovací systém musí být schopen vhodně a rychle reagovat na měnící se požadavky klienta. Ideální případ je reakce na změněné požadavky bez nutnosti úpravy zdrojového kódu, ale pouze úpravou konfigurace systému.

Na začátku procesu tvorby testeru není zcela jasné, jak bude finální produkt vypadat. Jak klient, tak programátor má hrubou představu toho, co se má blížit k finálnímu produktu, ale zcela jasné všechny operace nejsou. Jedná-li se o komplexní systém není možné přemýšlet o všech možných situacích, které mohou nastat. Je velmi vhodné, aby byl systém dostatečně flexibilní tak, aby na doplňující požadavky mohl reagovat bez nutnosti velkých zásahů. Nemusí se přímo jednat o změnu požadavků, jako spíše o doplnění požadavků. Čím více se systém bude blížit své finální podobě, tím více a přesněji bude možné o systému přemýšlet a vyplynou najevo věci, které je nutné změnit, upravit, nebo doplnit. Jedná se o úpravy, které budou nastávat během vývoje a vývojový tým bude schopen na takové požadavky reagovat.

Změna testovací sekvence může být motivována nalezením kritické chyby ve výrobku nebo prostým zavedením nové varianty do výroby. Při návrhu testovacího systému je vhodné myslet na to, že i za několik let může přijít požadavek na pozměnění testovací sekvence. Systém navržený flexibilně bude na neočekávané požadavky mnohem lépe reagovat.

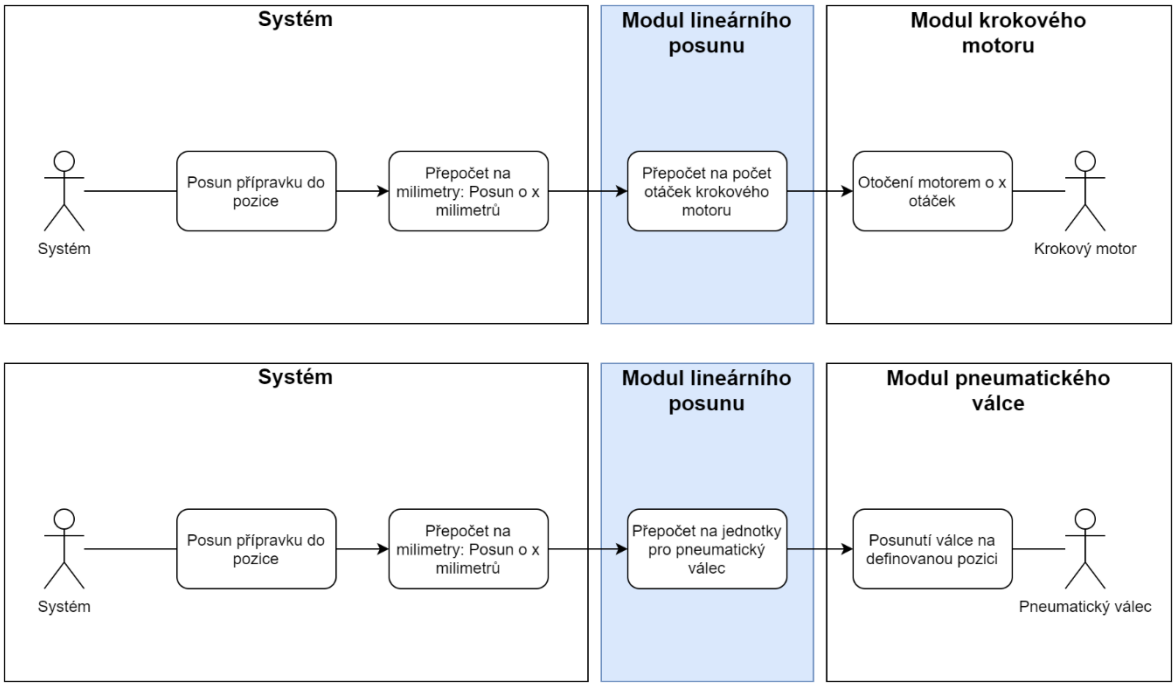
1.5. Znovu-použitelnost

Při vývoji více testovacích aplikací bude docházet k opakování velkého množství kroků mezi nimi. Struktura aplikací může být velice podobná. Některé testy se mohou opakovat. Komunikace s univerzálním HW bude jistě stejná.

Při špatném návrhu systému dojde k znovu-opisování stejného kódu, a proto je vhodné již od začátku myslet na to, že se kód bude znovu používat, třeba ve zcela jiném kontextu. Celý testovací systém by měl být navržen modulárně tak, aby některé, nebo dokonce všechny, části systému mohly být opakovaně použity.

Znovu-použitelnost vede k větší abstrakci. Více abstrakce ale znamená více implementace. Ideální je nalézt kompromis mezi praktičností a abstrakcí tak, aby se modul dal znovu použít, ale aby implementace netrvala příliš dlouho.

Klíčem k dosažení znovu-použitelnosti je rozdělení na moduly, izolování závislostí a komunikace na nezávisle definovaných jednotkách. Nevhodný příklad návrhu: Systém lineárního posunu s krokovým motorem, kde modul krokového motoru přijímá informaci o vzdálenosti a přepočítává ji na počet kroků, nebo počet otáček. Vznikne-li požadavek na vytvoření nového systému opět s krokovým motorem, nyní ale pro otáčení stolu. Nebude již vytvořený modul vyhovující. Vhodnější je vytvořit modul krokového motoru tak, aby přijímal informaci o počtu kroků, nebo počtu otáček. Je zodpovědností systému přepočítat vzdálenost na počet otáček. V případě změny v systému a nahrazení krokového motoru za pneumatický válec bude opět jen zodpovědnost systému přepočítat jednotky a výměna modulů proběhne snadno. Definování přepočtu jednotek mimo zdrojový kód aplikace je ideálním případem. Dojde-li ke změně požadavků, bude možné změnit strukturu systému bez nutnosti opětovně sestavovat spustitelnou aplikaci. Na obr. 1 je zobrazen návrh systému s modulem lineárního posunu.



Obr. 1 — Modularita: návrh modulu lineárního posunu

1.6. Automatický tester, Průmysl 4.0

Trendem ve vývoji HW jsou konfigurovatelná zařízení oproti zařízením jednoúčelovým. Dochází k oddělení HW a SW. HW je více univerzální a je možné jej flexibilně nakonfigurovat právě pomocí SW. Je možné využít jeden stejný HW pro více druhů aplikací. Tato úroveň znovu-použitelnosti lze využít i při výrobě jednoho výrobku, kde je HW nakonfigurován pro některé testy určitým způsobem a pro jiné testy způsobem jiným. Změnu konfigurace je možné provést v průběhu testování. Stejný trend by měl být sledován při vývoji softwarových nástrojů pro práci s tímto HW.

Jedním z konceptů Průmyslu 4.0 jsou výrobní buňky. Jsou to samostatné celky, které jsou schopné provádět více různých druhů operací a měnit své chování na základě požadavků výrobku. Výrobní jednotka neurčuje, jaké budou provedeny operace, ale pouze nabízí své služby. Výrobek si sám určí, jaké operace budou provedeny případně, které služby budou využity. Je to v souladu s konfigurovatelným HW, oproti jednoúčelovému. Výrobní jednotka má jeden HW pro mnoho svých aplikací a jeho chování se mění změnou SW. Znovu-použitelnost systému lze povznést na vyšší úroveň pomocí znovu-použití stejného SW. Změna chování SW se provede změnou jeho konfigurace, která není součástí kódu, ale je nezávislá a oddělená.

Celá funkcionalita systému může být definována v konfiguraci, pokud je systém navržen správným způsobem. Vhodným řešením je vývoj vlastního skriptovacího jazyka s možností jeho interpretace v testovací/výrobní znovu-použitelné aplikaci. Takový jazyk musí být schopen provádět rozhodovací operace a mít přístup k celému systému. Celá testovací sekvence se bude načítat jako vstupní informace systému. Změnu požadavků na systém je možné provést změnou ve skriptech bez nutnosti zasahovat do zdrojového kódu aplikace. Změna chování výrobní jednotky/univerzálního testeru lze realizovat právě pouhou změnou konfigurace. Údržba a vývoj SW je jednodušší a levnější pro jednu verzi aplikace, která je univerzální oproti mnoha verzím.

Dle [4] by měly být průmyslové automatizační systémy schopné zvýšit softwarovou univerzálnost zavedením flexibilní konfigurace nejrůznějších systémových parametrů. Principy a hlavní funkce průmyslových automatizačních systémů mohou zůstat nezměněné pro různé průmyslové aplikace. Detaily pro jakoukoli specifickou aplikaci mohou být předefinovány modifikací konfigurační databáze dle jakýchkoli specifických uživatelských požadavků. Kombinací konfigurační databáze a fixních systémových modulů je dosaženo specifické konfigurace aplikace, a tím je postaven průmyslový automatizační software s požadovanou funkcionalitou.

1.7. Problémy s textově zobrazovanými testovacími sekvencemi

Jelikož je pro automatický tester definovaná vlastní syntaxe jazyka, je pravděpodobné, že daný jazyk nebude mít podporu ve vývojových prostředích a bude nutné jej psát v textovém editoru. V textovém editoru není implementovaná kontrola parity závorek, kontrola zanořování, automatická korekce syntaxe a další funkce. V čistém neformátovaném textu je snadné se ztratit nebo přehlédnout překlep. Graficky nejsou odděleny struktury, a tak může být textově zobrazený kód nepřehledný.

Řešením je vytvoření interaktivní aplikace pro práci s těmito skriptovanými testovacími sekvencemi. Taková aplikace bude graficky rozlišovat jednotlivé příkazy a jednoznačně vymezovat struktury. Komentáře by měly být zobrazeny odlišně tak, aby nebraly pozornost od výkonného kódu. Důležitou vlastností je možnost otevření více souborů a přepínání mezi nimi pro lepší organizaci práce. Tato interaktivní aplikace musí být navržena dostatečně flexibilně proto, aby bylo možné držet krok s vyvíjejícím se skriptovacím jazykem případně, aby existovala možnost rozšíření pro nové skriptovací jazyky.

2. Popis problematiky objektově orientovaného programování, grafického vývojového prostředí a základní syntaxe jazyka použitého pro informační měřící systém

Pro návrh velkých aplikací je vhodné používat objektově orientované programování pro jeho značné výhody v oblasti rozdělování komplexního chování na několik malých jednoduchých pochopitelných částí. Aplikace bude v budoucnu integrovaná do automatizovaného měřicího systému a z toho důvodu musí být napsaná v LabVIEW. Jedná se o programovací jazyk, ale také o vývojové prostředí. Kód je zobrazen graficky formou blokového diagramu. V této kapitole je popsán skriptovací jazyk automatického testeru ATEsteru, pro který bude aplikace vytvořena.

2.1. Objektově orientované programování

Objektově orientované programování je jedno z programovacích paradigmat, které nahlíží na části software jako na objekty. Objekty jsou definovány dvěma komponenty, a to atributy a chováním, právě přítomnost obou těchto komponent je hlavní rozdíl mezi OOP a ostatními paradigmaty[5]. OOP není pro lidi nový koncept, protože lidé přemýšlejí v objektech přirozeně. Například auto má své atributy: kola, volant, kapota atd. a své chování: může jezdit, zastavovat, otevírat dveře atd. Pro programátory se jedná o usnadnění programování v tom smyslu, že programátor přemýšlí tak, jak je pro něj přirozené a nezabíhá do přílišných detailů tam, kde to není nutné. Objekt z hlediska programu lze chápat jako balíček zahrnující jak data, tak funkce pro práci s těmito daty.

Hlavními objektově orientovanými koncepty jsou: zapouzdření, dědičnost, polymorfismus a kompozice[5].

2.1.1. Třída a objekt

Třída je statickým předpisem svých objektů. Znamená to, že třída udává předpis toho, jak budou objekty vypadat. Může být chápána jako šablona, podle které se objekty vytvářejí. Příkladem mohou být třídy: auto, telefon, počítač. Jedná se o předpisy. Auto má kola a může jezdit, jaká ale kola má už je specifické pro konkrétní auto, tedy objekt.

Objekty jsou instance, datové reprezentace, tříd. Jsou to entity, které jsou vytvořeny na základě svých tříd, předpisů, šablon. Příkladem je konkrétní auto zaparkované na parkovišti, telefon ležící na stole, určitý existující počítač. Rozdíl je v tom, že o třídě auto je možné pouze přemýšlet, ale fyzicky neexistuje, zatímco s objektem auto již můžeme pracovat, jelikož se jedná o skutečnou věc.

Třída jako předpis je pouze jedna, ale objektů podle ní může být vytvořeno mnoho. Stejně jako existuje jedna obecná představa o tom, co je auto a jak s ním pracovat, ale skutečných aut vyrobených podle tohoto předpisu je mnoho.

2.1.2. Zapouzdření

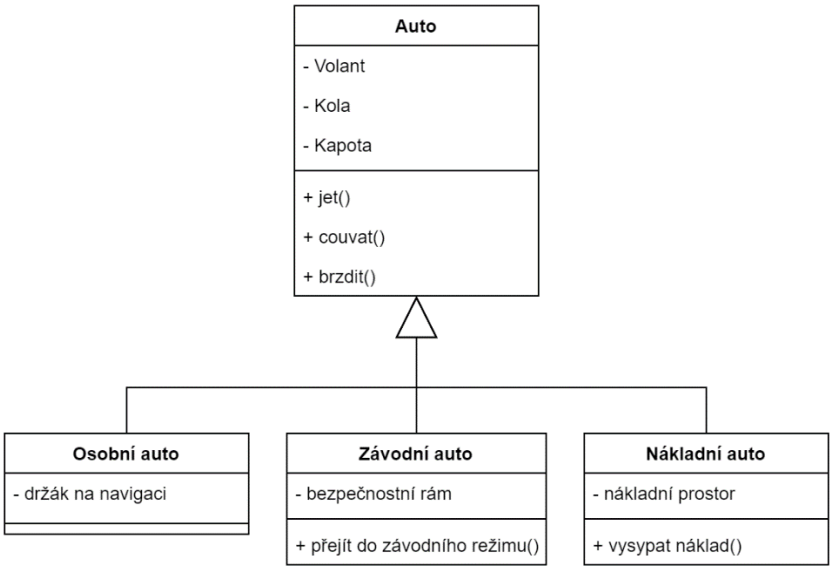
Jednou z hlavních výhod používání objektů je, že objekty nepotřebují odhalovat všechny své atributy a chování [5]. Jinými slovy někdy je vhodné některé atributy a chování skrývat. Je to dobré pro to, aby byl omezen rozsah přemýšlení. Například u kalkulačky je vhodné skrývat algoritmy použité pro výpočty, protože uživatel nepotřebuje pro vypočtení výsledku znát algoritmus, který výpočet provedl. Jediné,

co uživatel potřebuje je výsledek. Díky tomuto se rozdělí komplexnost systému na malé části, se kterými se dá rozumně pracovat.

Zapouzdření je také vhodné z hlediska bezpečnosti. Některé části objektů není vhodné uživateli poskytnout, například databázi s přístupovými údaji.

2.1.3. Dědičnost

Dědičnost v OOP je koncept, který dovoluje velmi rychle a jednoduše změnit chování, nebo rozšířit již existující třídy. Příklad z reálného světa je třída Auto, které má kola, dveře, volant atd. potomek této třídy je Nákladní auto, které dědí vše, co měl její předek a přidává něco navíc. Nákladní auto má stejně jako obyčejné Auto volant, kola, dveře, ale navíc má například nákladní prostor. O všech nákladních autech můžeme říct, že se jedná o auta, ale ne o všech autech můžeme říct, že se jedná o nákladní auta. Dalším potomkem třídy auto může být závodní auto, viz obr. 2.



Obr. 2 — OOP: hierarchie dědičnosti třídy auto

2.1.4. Polymorfismus

Třída může mít mnoho potomků, kteří mohou dále mít své potomky atd. Polymorfismus je koncept, který umožňuje použít potomka na místě svého předka. Potomek může nahradit svého předka. Z pohledu uživatele třídy nezáleží na tom, který potomek je používán, protože každý z nich obsahuje vše, co měl jeho předek. Takovému nahrazení předka za potomka se říká polymorfní přiřazení a je v OOP hojně používáno. Díky tomuto konceptu je možné vytvářet obecné předky, kteří sdružují společné chování.

Na parkovišti mohou stát Auta a nezáleží na tom, jestli jsou to Závodní auta, Osobní auta, nebo Nákladní auta. Jediné, co parkoviště zajímá je to, že se jedná o Auta.

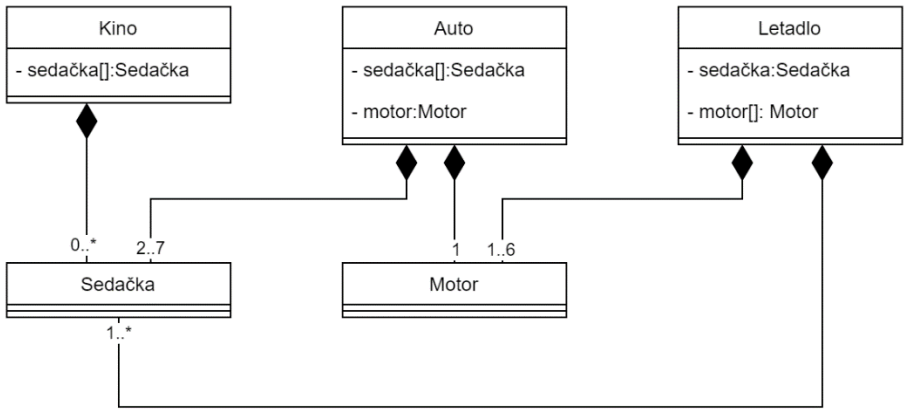
Tento princip nefunguje opačně. Není možné nahradit potomka jeho předkem, protože předek neobsahuje vše, co obsahují jeho potomci. Například v závodu mohou odstartovat pouze Závodní auta. Předek obecné Auto neobsahuje vše, co Závodní auto, a proto není do závodu připuštěn.

Tento koncept se dá výhodně využít pro automatické testery, kde definujeme společného předka Voltmetr a dědičné třídy: Měřící karta v PC, Ruční voltmetr a Analogový voltmetr. Pro dosažení požadované funkcionality z pohledu systému nemusí být důležitá informace o jaký konkrétní typ voltmetru se jedná, dokud dokáže změřit napětí.

2.1.5. Kompozice

Ne každý objekt musí být složený z elementárních částí. Je vhodné, aby objekt mohl být složený z jiných objektů. V příkladu třídy Auto, je rozumné chápat motor jako samostatný objekt. Auto je tedy objektem a skládá se z několika dílčích objektů. Na obr. 3 je zobrazeno, že auto se skládá ze tříd: Motor a několika Sedaček, což jsou samostatné objekty. Tyto objekty, ale nejsou vyhrazené pouze pro Auto. Kino má mnoho Sedaček. Letadlo má také mnoho Sedaček a má několik Motorů. Kompozicí tedy rozumíme skládání objektů do nových celků, bez nutnosti opisování stejného kódu.

Tento koncept je vhodný pro dosažení znovu-použitelnosti, kde můžeme definovat několik obecnějších objektů, ze kterých se budou skládat komplexnější objekty. Usnadnění je v tom, že není nutné znovu programovat funkcionalitu specifickým objektům, ale pouze ji použít jako součást něčeho složitějšího.



Obr. 3 — OOP: kompozice

2.1.6. Abstraktní třída

Jedná se o speciální typ třídy, která zahrnuje abstrakci. Příkladem může být zoo, která pracuje se zvířaty a s ošetřovateli. Z nějakého důvodu může být nutné definovat společného předka jak pro zvířata, tak pro ošetřovatele. Nabízí se vytvořit abstraktní třídu Živá bytost, ze které budou jak Zvířata, tak Ošetřovatelé dědit. Abstraktní třída slouží k tomu, aby definovala společné chování, ale ne, aby všechno chování implementovala. Protože Zvíře i Ošetřovatel mají jméno, při vytváření samostatných tříd pro Ošetřovatele i pro Zvíře, by bylo nutné napsat metody pro uložení jména a vyčtení jména do obou tříd a tím znovu opisovat stejný kód. Je vhodné, toto společné chování vyjmout a implementovat na pouze jednom místě, a to v abstraktní třídě Živá bytost, která bude mít atribut jméno a metody pro zobrazení jména. Zoo poté může pracovat se všemi Živými bytostmi, v tomto případě vypsat jména všech Živých bytostí v Zoo. Abstraktní třída definuje chování, ale nemusí jej nutně implementovat. Předepsané, ale ne implementované, metody se nazývají abstraktní metody. Abstraktní třídy jsou normální třídy, které mají alespoň jednu abstraktní metodu a udávají svým potomkům povinnost implementovat všechny abstraktní metody. Pro příklad Živé bytosti může být abstraktní metoda

„vydejZvuk“(). Nemá smysl, aby abstraktní Živá bytost vydávala nějaké zvuky, protože se nejedná o konkrétní Živou bytost, ale pouze o popis společného chování různých tříd. Abstraktní třída v tomto případě udává povinnost svým potomkům implementovat metodu „vydejZvuk()“ a každý potomek této třídy, tedy Zvíře a Člověk bude tuto metodu implementovat.

2.1.7. Interface

Rozšíření abstraktní třídy je interface neboli čistě abstraktní třída. Jedná se o normální třídu, která má všechny metody abstraktní. Příklad interface může být „MáHmotnost“, který nebude mít žádná data, ani implementované metody, jediné, co bude mít, je abstraktní metoda „získatHmotnost()“, kterou budou muset její potomci implementovat. Každého potomka, který z této třídy bude dědit, bude možné zvážit.

Interface nevnucuje potomkům konkrétní implementaci, ale vyžaduje jejich vlastní implementaci. Výhodou používání interface je, že uživatel může pracovat s objekty, které jsou naprosto odlišné, ale zacházet s nimi stejně. Třídy mohou implementovat více interface, a tak je možné pracovat s třídou poprvé skrze jeden interface, podruhé skrze jiný interface.

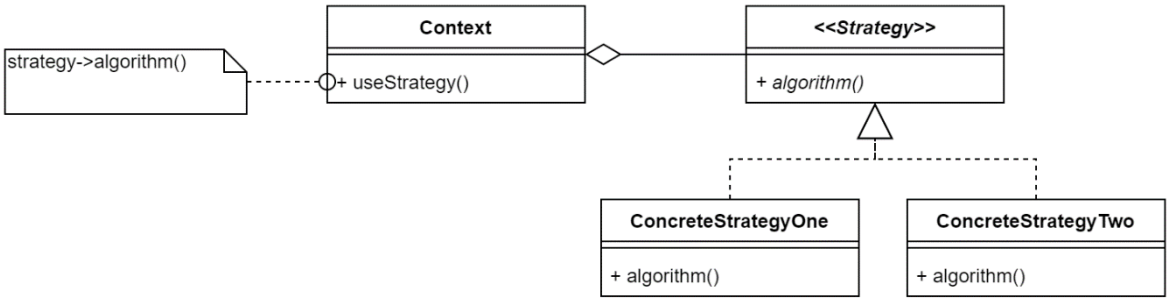
2.1.8. Návrhové vzory

Návrhový vzor pojmenovává a identifikuje klíčové aspekty podobných návrhů struktur, které jsou užitečné pro vytváření znovu-použitelného objektově orientovaného designu. Návrhový vzor identifikuje zúčastněné třídy a objekty, jejich role, spolupráci a rozdělení zodpovědností. Každý návrhový vzor se soustředí na specifický návrhový problém. Popisuje, kdy lze vzor použít, jestli může být použit z pohledu jiných návrhových omezení, důsledky a kompromisy jeho použití.[6]

2.1.8.1. Návrhový vzor – Strategy (Strategie)

Pokud izolujeme odlišné části od zbytku kódu, bude později možné změnit nebo rozšířit oddělené části bez zásahu do původního systému. Výsledkem je méně nechtěných důsledků ze změn kódu a více flexibility v systému.[7]

Na obr. 4 vidíme, že místo toho, aby se ve třídě „Context“ používala přímo konkrétní strategie a implementoval se specifický algoritmus, je mnohem vhodnější vyjmout tuto strategii a vytvořit interface „Strategy“, konkrétní strategie potom budou implementovat tento interface. Důsledek je, že je možné rozšířit nebo změnit specifický systém vložením jiné strategie, bez nutnosti jakkoli měnit kód systému. Jediné, co se změní je hierarchie „Strategy“. Strategie může být jakékoli chování systému. Například zvířeti může být vloženo chování „Chodit“ a „Jíst“. Konkrétní zvířata pak mohou měnit své chování vkládáním jiných instancí chování. Zvíře, pokud je mladé, bude pravděpodobně jíst jiným způsobem než zvíře, které je dospělé. Tento návrhový vzor nám umožní dynamicky měnit chování objektu bez nutnosti měnit původní objekt, případně bez nutnosti vytvářet potomky pro všechny kombinace různého chování.

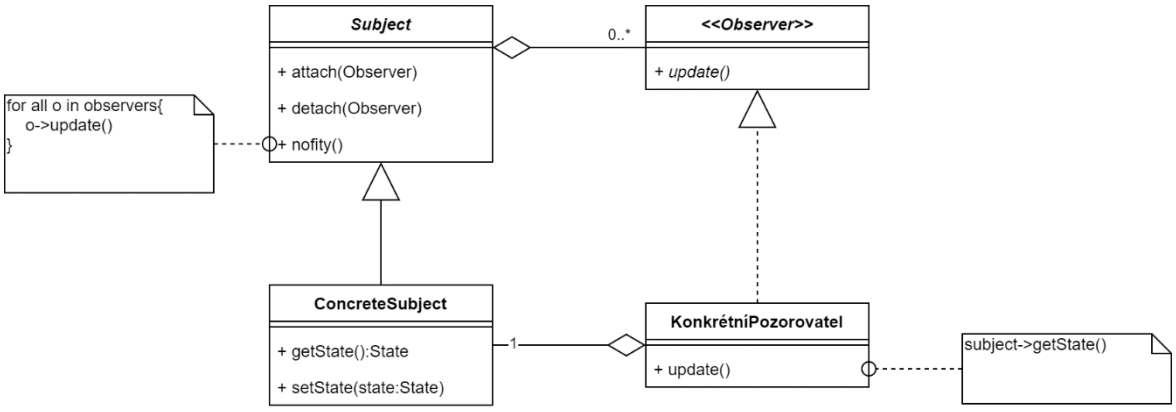


Obr. 4 — OOP: Návrhový vzor – Strategy

2.1.8.2. Návrhový vzor – Observer (Pozorovatel)

Návrhový vzor Observer definuje závislostní vztah mezi objekty jeden ku mnoho tak, že když objekt změní svůj stav, všechny jeho závislosti o tom jsou informovány a aktualizovány automaticky. [7]

Ve chvíli, kdy se změní stav konkrétního subjektu, všichni pozorovatelé, kteří požádali o sledování stavu, jsou o změně informováni. Jakmile je pozorovatel o změně informován, je na něm, zda si vyzvedne informaci o novém stavu z konkrétního subjektu. Alternativou je dotazování na změnu stavu (tzv. polling), kde všichni pozorovatelé budou pravidelně periodicky sledovat stav subjektu a vyhodnocovat, jestli nedošlo ke změně. V takovém případě bude docházet ke zbytečnému zatěžování komunikačního média a může se stát, že frekvence dotazování nebude dostatečně velká a pozorovatel nezachytí všechny změny stavu. Pokud je použit návrhový vzor Observer, k těmto chybám a nedostatkům docházet nebude, protože k zatížení komunikačního média dojde pouze ve chvíli, kdy se změní stav subjektu. Pozorovatelé tak budou informováni o všech změnách. Na obr. 5 je zobrazena implementace tohoto návrhového vzoru. Jak je zobrazeno, pozorovatelé mohou být přidáváni a ubíráni za běhu. Jakmile v konkrétním subjektu vznikne změna stavu, je zavolána metoda předka „notify()“, která propaguje informaci o změně stavu všem svým pozorovatelům a na nich volá metodu „update()“, ve které si pozorovatel může vyzvednout informaci o novém stavu subjektu.



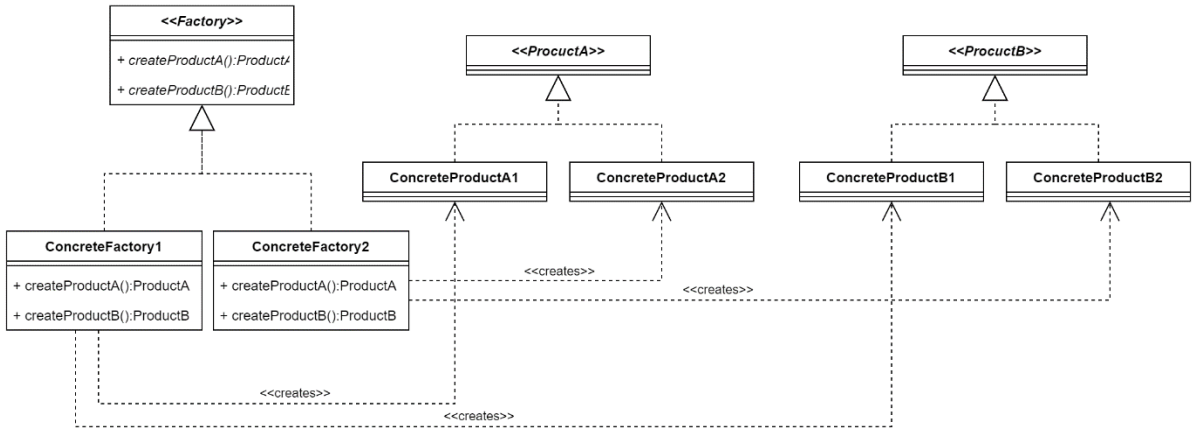
Obr. 5 — OOP: Návrhový vzor – Observer

Hlídání kritické hodnoty je typická aplikace, kde lze s velkou výhodou tento návrhový vzor využít. Je jednodušší implementovat hlídání kritické hodnoty v místě, kde se data získávají a později o změně stavu informovat všechny, které změna zajímá.

2.1.8.3. Návrhový vzor – Abstract Factory (Abstraktní továrna)

Návrhový vzor Abstract Factory poskytuje rozhraní pro vytváření rodin podobných nebo závislých objektů bez specifikování jejich konkrétních tříd. [7]

Z obr. 6 je patrné, že „ConcreteFactory1“ vytváří „ConcreteProductA1“ a „ConcreteProductB1“ a obdobně pro „ConcreteFactory2“. Pomocí tohoto návrhového vzoru je umožněno systému vzdát se zodpovědnosti za vytváření objektů. Komplexnost pro vytvoření a inicializaci složitých objektů je uzavřena do továrny. Z pohledu systému nemusí být potřebné vědět, jaký konkrétní produkt je vytvářen a jak, ale že je produkt z hierarchie požadovaného produktu. Například pokud je požadavek na to mít v systému světlý a tmavý mód, je vhodné vytvořit TmavýMódFactory a SvětlýMódFactory, které budou vytvářet konkrétní produkty. Systém požaduje, aby bylo vytvořeno Tlačítko a umístěno na určitou pozici, ale jestli se bude jednat o Tmavé tlačítko, nebo Světlé tlačítko již není důležité, tato zodpovědnost je přenechána na TmavýMódFactory a SvětlýMódFactory. Uživatel ovlivní tvorbu konkrétních produktů výměnou konkrétní továrny.



Obr. 6 — OOP: Abstract factory design pattern

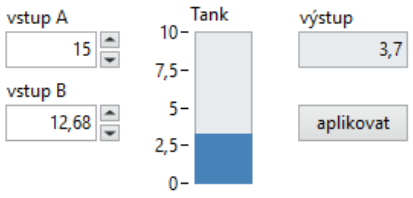
2.2. Grafické vývojové prostředí LabVIEW

LabVIEW je jak programovací jazyk, tak vývojové prostředí. Je vyvíjeno společností NI, dříve National Instruments, od roku 1983. Jedná se o grafický programovací jazyk. Kód v tomto jazyce připomíná schéma a jednotlivé hodnoty a signály jsou zobrazeny symbolicky pomocí drátů, které jsou zapojovány do ikon. Někdy je jazyk LabVIEW nazýván G-jazyk, což je zkratka pro grafický jazyk.[9]

Kód LabVIEW se skládá ze dvou hlavních částí, a to čelní panel a blokový diagram.

2.2.1. Čelní panel

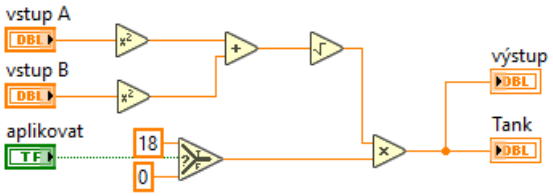
Čelní panel slouží jako rozhraní pro uživatele. Jsou na něm zobrazovány jak výstupní, tak výstupní prvky. Příkladem prvků čelního panelu mohou být grafy, textová pole, číselná pole, tlačítka atd. Není nutné je složitě vytvářet, ale jsou k dispozici z paletek. Prvky se rozmisťují na čelní panel přetažením myší. Vývoj čelního panelu je rychlý a intuitivní. Na obr. 7 je zobrazena ukázka čelního panelu.



Obr. 7 — LabVIEW: ukázka čelního panelu

2.2.2. Blokový diagram

Blokový diagram slouží k tvorbě kódu pomocí ikon, které jsou propojené dráty. Propojení mezi čelním panelem a blokovým diagramem je realizováno pomocí speciálních ikon v blokovém diagramu zvaných terminály. Programátor zobrazí hodnoty na čelním panelu připojením drátů s hodnotami do terminálů grafických prvků čelního panelu a hodnoty jsou automaticky zobrazeny. Po sestavení a spuštění sestavené aplikace jsou uživateli zobrazeny pouze čelní panely a blokové diagramy jsou skryty. Příklad blokového diagramu je zobrazen na obr. 8.



Obr. 8 — LabVIEW: ukázka blokového diagramu

2.2.3. Výhody grafického programování

Díky grafické reprezentaci vazeb mezi jednotlivými bloky je na první pohled zcela zřejmé odkud data přicházejí a kam jsou připojena, jak je vidět na obr. 8. V textově orientovaných jazycích je stejná funkcionalita zajištěna pojmenováním proměnné, případně využitím návratové hodnoty. Na obr. 9 je napsán stejný kód jako na obr. 8 ale v jazyce C++.

```
int main(int arg, char** args)
{
    double vstupA, vstupB, vystup, tank;
    bool aplikovat;
    cin >> vstupA;
    cin >> vstupB;
    cin >> aplikovat;
    vstupA = pow(vstupA, 2);
    vstupB = pow(vstupB, 2);
    double temp = sqrt(vstupA + vstupB);
    if (aplikovat)
    {
        temp = temp * 18;
    }
    else
    {
        temp = temp * 0;
    }
    vystup = temp;
    tank = temp;
    cout << "Vystup: " << vystup << " Tank: " << tank << endl;
    return 0;
}
```

Obr. 9 — LabVIEW: porovnání s textovým zobrazením

Grafická tvorba kódu je intuitivní. V reálném světě se pracuje převážně s fyzickými objekty a většina informací je zpracována vizuálně. Mnohdy je důležité si text představit v grafické podobě, aby dával smysl. Při programování graficky není nutné si nic představovat, protože grafická forma je již zobrazena. Hlavní výhodou je možnost sledování cesty dat, což v textovém zobrazení probíhá hledáním deklarace nebo definice proměnné.

2.2.4. Nevýhody grafického programování

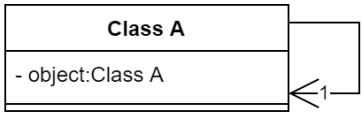
Programování v grafické orientovaném jazyce může v některých případech trvat déle než v textově orientovaných jazycích. Programátor stráví nemalou část času připojováním a zarovnáváním drátů, což v textově orientovaných jazycích programátor dělat nemusí. Proměnné jednoduše zapíše a nemusí se zabývat připojováním drátu ze zdrojového místa.

2.2.5. Podpora OOP v LabVIEW

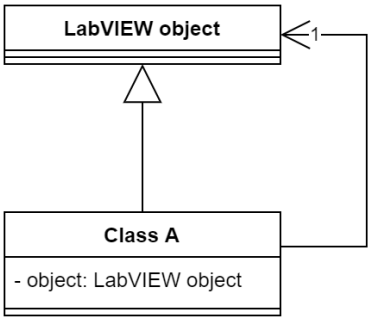
LabVIEW podporuje objektově orientované programování. Podporuje dědičnost, polymorfismus, zapouzdření i kompozici objektů. Od verze LabVIEW 2020 je implementována podpora interface[10]. Každá třída je potomkem třídy „LV Object“. Všechny objekty je tedy možné skladovat v polymorfní datové struktuře tohoto typu.

LabVIEW podporuje pouze privátní datové položky, nepodporuje veřejné a protected. Pro přístup k datovým položkám mimo objekt, je nutné vytvořit přístupové metody a těm přiřadit příslušná práva.

Při práci s třídami v LabVIEW existuje několik omezení. Třída nesmí v privátních datech obsahovat objekt stejné třídy, ani referenci na objekt stejné třídy, viz obr. 10. Řešením tohoto problému je vložení reference na objekt třídy „LV Object“, nebo jakékoli předchozí třídy v dědičné hierarchii a přiřadit do proměnné třídu požadovaného, stejného typu, viz obr. 11. Při práci s tímto objektem stačí, když se pokaždé přetypuje „LV Object“ na typ požadované třídy.

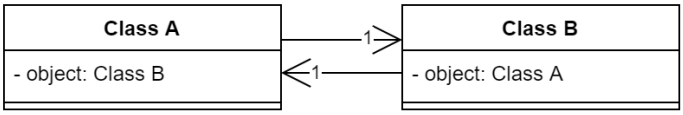


Obr. 10 — LabVIEW: restrikce objektu stejné třídy

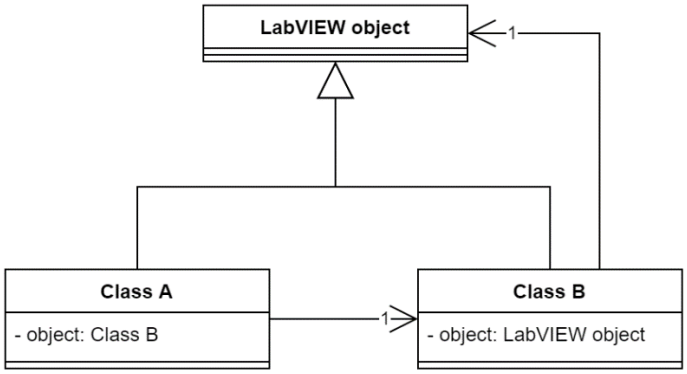


Obr. 11 — LabVIEW: řešení restrikce objektu stejné třídy

Třída nesmí v privátních datech obsahovat objekt, nebo referenci na objekt třídy, která ve svých privátních datech obsahuje objekt, nebo referenci na objekt této třídy, viz obr. 12. Toto omezení se opět dá obejít vložením alespoň do jedné ze dvou tříd objekt typu „LV Object“ a při práci s tímto objektem použít přetypování na cílovou třídu, viz obr. 13.



Obr. 12 — LabVIEW: restrikce objektu třídy s objektem této třídy



Obr. 13 — LabVIEW: řešení restrikce objektu třídy s objektem této třídy

2.3. Syntaxe skriptovacího jazyka

Dále je rozebrána syntaxe skriptovacího jazyka univerzálního testeru ATEster. Jedná se o textově orientovaný jazyk, který je interpretován za běhu aplikace. Jazyk se zapisuje do textových souborů s příponou „.script“ a „.macros“.

2.3.1. Proměnné, datové typy, komentáře

Deklarace proměnné se provádí na samostatném řádku. Proměnná je uvozena symbolem dolar ($). Příkaz je ukončen novým řádkem (\n). Deklarace se provádí zápisem datového typu a názvu proměnné, viz první řádek na obr. 14. Datové typy jazyka jsou popsány v tab. 1.

Definice se provádí zapsáním názvu proměnné, symbolu rovná se (=) a názvu nové hodnoty, nebo nové proměnné, viz druhý řádek na obr. 14.

```
str $nazevPromenne
$nazevPromenne = hodnota promenne
```

Obr. 14 — Syntaxe jazyka: deklarace a definice proměnné

Lze spojit deklaraci s definicí. Poté lze psát datový typ proměnné, název proměnné, symbol rovná se a hodnota, nebo název jiné proměnné, viz obr. 15.

```
sgl $prvniPromenna = 16.2
sgl $druhaPromenna = $prvniPromenna
```

Obr. 15 — Syntaxe jazyka: spojená deklarace a definice

Tab. 1 — Syntaxe jazyka: datové typy

Datový typ	Příklad použití	Popis
sgl	sgl $delka = 3,15	datový typ pro ukládání desetinných čísel
str	str $jmeno = text	datový typ pro ukládání textových řetězců
int	int $hodnota = 7	datový typ pro ukládání celých čísel
bool	bool $ulozeno = True	datový typ pro ukládání logických hodnot

Jazyk podporuje základní matematické operace zobrazené v tab. 2. Matematické operace lze řadit za sebe. Při zařazení více matematických operací se provádějí zleva doprava. Žádná operace nemá přednost před jinou. Použití je znázorněno na obr. 16.

```
sgl $promenna
sgl $dalsiPromenna
$promenna = $promenna + 10 + $dalsiPromenna * 2
```

Obr. 16 — Syntaxe jazyka: matematické operace

Tab. 2 — Syntaxe jazyka: matematické operace

Symbol	Operace	Příklad použití	Popis
+	součet	$poradi = $poradi + 10	provede součet dvou proměnných (nebo konstant) a přiřadí do proměnné
-	rozdíl	$poradi = $poradi - 10	provede rozdíl dvou proměnných (nebo konstant) a přiřadí do proměnné

*	součin	$poradi = $poradi * 10	provede součin dvou proměnných (nebo konstant) a přiřadí do proměnné
/	podíl	$poradi = $poradi / 10	provede podíl dvou proměnných (nebo konstant) a přiřadí do proměnné
++	inkrementace	$poradi++	zvýší hodnotu proměnné o 1
--	dekrementace	$poradi--	sníží hodnotu proměnné o 1

Ve skriptech je vše, co je uvedeno za symbolem středník považováno za komentář.

2.3.2. Makra

Jazyk má možnost vytvoření make. Makro je blok kódu, který je označen názvem. Tento blok kódu je poté při načtení skriptu překopírován do místa volání. Makro je možné zavolat z více míst. Jednotlivá makra se nepíšou do skriptových souborů, ale vytvářejí se pro ně samostatné soubory s příponou „.macros“.

Definice makra se provádí zápisem klíčového slova „macro“ symbol hashtag (#) a jméno makra. Pro vložení parametru se za názvem makra napíše čárka (,) a název parametru. Pro přidání více parametrů se za název makra napíše čárka (,) a názvy všech parametrů oddělených čárkami. Příklad definice je uveden na obr. 17. Blok kódu, který se vykoná při zavolání makra je uzavřen mezi složené závorky.

```
; definice makra
macro #zmerit_napeti,svorka_1, svorka_2
{
        ;změření napětí mezi svorkami
}
```

Obr. 17 — Syntaxe jazyka: definice makra

Pro volání makra je nutné napsat klíčové slovo „macro“ a do kulatých závorek uvést název makra. Pro předání parametrů makru se v závorkách za název uvedou parametry oddělené čárkou, viz obr. 18.

```
;zavolání makra
macro(zmerit_napeti,$cislo_svorky_1,$cislo_svorky_2)
```

Obr. 18 — Syntaxe jazyka: volání makra

2.3.3. Linkování

Aby bylo možné zavolat makro, je nutné, aby byl do skriptu nalinkován soubor s jeho definicí. K tomu slouží příkaz „define“ a do kulatých závorek se napíše absolutní nebo relativní cesta k souboru s makry. Na obr. 19 je uveden příklad linkování souboru s makry do hlavního skriptu.

```
;nalinkování souboru s makry
define(\SlozkaMaker\rizeni_motoru.macros)
```

Obr. 19 — Syntaxe jazyka: linkování maker

Jazyk dovoluje rozdělit skript na více samostatných skriptů a volat je mezi sebou. Pro zavolání a provedení skriptu z jiného místa slouží příkaz „include“ a do kulatých závorek se uvede cesta ke skriptu.

```
;nalinkování jiného skriptu
include(\Slozka_dilcich_skriptu\jiny_script.script)
```

Obr. 20 — Syntaxe jazyka: linkování skriptu

2.3.4. Podmínky

Je implementována podpora větvení pomocí příkazu „if“. Za příkazem následuje symbol dolar ($) a podmínka. Pokud je podmínka vyhodnocena jako „True“ je proveden následující blok příkazů. Je-li ale podmínka vyhodnocena jako „False“, provede se blok příkazů za „else“, který nemusí být uveden. Příkaz „if“ vyžaduje jako následující symbol dolar. Jelikož proměnná má jako první symbol dolar zkracuje se zápis na pouze jeden tento symbol, jak je zobrazeno na obr. 21.

```
if $promenna == False
{
        ;kladná větev
}
else
{
        ;záporná větev
}
```

Obr. 21 — Syntaxe jazyka: větvení

Podmínka příkazu může být přímo hodnota typu bool, proměnná typu bool, porovnání pomocí porovnávacích operátoru, nebo kombinace předešlých s použitím logických operátorů.

Porovnání je možné provést pomocí relačních operátorů, které jsou uvedeny v tab. 3. Operátor vyhodnotí operaci dvou operandů, které operaci obklopují, jak je uvedeno na obr. 22.

```
str $promenna = TextovaHodnota
if $promenna == TextovaHodnota
{
        ;podmínka je vyhodnocena jako "True"
}
```

Obr. 22 — Syntaxe jazyka: operátor „rovná se“

Tab. 3 — Syntaxe jazyka: operátory porovnání

Operátor	Porovnání	Popis
==	rovná se	vrací „True“, pokud jsou hodnoty stejné
!=	nerovná se	vrací „True“, pokud jsou hodnoty rozdílné
<	menší než	vrací „True“, pokud je první operand menší než druhý operand
>	větší než	vrací „True“, pokud je první operand větší než druhý operand

Ve skriptech lze skládat více podmínek dohromady pomocí logických operátorů, které jsou uvedeny v tab. 4. Při kombinování více než jednoho logického operátoru se provede nejdříve operátor „logický součin“ a poté se provede „logický součet“. Pro změnu pořadí je nutné uzavřít podmínky, které se mají vyhodnotit samostatně do kulatých závorek. Příklad použití je uveden na obr. 23.

```
str $promenna = TextovaHodnota
int $cislo = 15
bool $logicka_promenna = False
if $($promenna != TextovaHodnota || $cislo > 20) && $logicka_promenna
{
        ;kladná větev
}
```

Obr. 23 — Syntaxe jazyka: skládání podmínek

Tab. 4 — Syntaxe jazyka: logické operátory

Operátor	Porovnání	Popis
||	logický součet	vrací „True“, pokud je alespoň jeden z operandů „True“
&&	logický součin	vrací „True“, pokud jsou oba operandy „True“

Mnoha vnořených „if“ „else“ příkazů se dá nahradit příkazem „switch“. Zapisuje se klíčovým slovem „switch“, do závorek za příkaz se uvede skriptová proměnná. Následuje blok kódu uzavřený do složených závorek. Jednotlivé větve se uvozují klíčovým slovem „case“, za kterým následuje hodnota. Pokud se hodnota daného „case“ rovná aktuální hodnotě skriptové proměnné, vykoná se kód následující za tímto „case“. Pro ukončení běhu ve větvi se píše příkaz „break“. Pokud je nutné provést nějaké činnosti, i když se žádná hodnota „case“ nerovná hodnotě skriptové proměnné, je možné použít větev „case default“, která se vykoná, pokud žádná předchozí větev nebyla vykonána. Příklad použití je zobrazen na obr. 24.

```
switch($ciselna_promenna)
{
        case 0
                ;provede se, pokud je proměnná
                ;rovna nule
                break
        case 1 || 2
                ;provede se, pokud je proměnná
                ;rovna jedničce nebo dvojce
                ;pokud není uveden "break"
                ;provede se i následující case
        case 3
                break
        case default
                ;provede se, pokud se proměnná
                ;nerovná ani jedné z předchozích
                ;hodnot
                break
}
```

Obr. 24 — Syntaxe jazyka: příkaz switch

2.3.5. Cykly

Pro vykonání části kódu vícekrát jsou v jazyce implementovány dva druhy cyklů. První je cyklus typu „do while“. Tento cyklus provádí blok kódu, dokud platí definovaná podmínka. Blok kódu se nejprve provede a poté se vyhodnocuje podmínka. Syntaxe cyklu je následující: blok kódu uzavřený ve složených závorkách, klíčové slovo „while“ a v kulatých závorkách podmínka, jak je znázorněno na obr. 25.

```
int $prom = 0
{
        ;proměnná se může libovolně měnit
        $prom++
}while($prom < 15)
```

Obr. 25 — Syntaxe jazyka: cyklus „do while“

Druhý implementovaný cyklus je cyklus „for“. Tento cyklus provede blok kódu tolikrát, kolikrát definuje vstupní proměnná. Syntaxe je klíčové slovo „for“, v kulatých závorkách skriptová proměnná, nebo číslo specifikující počet opakování bloku kódu, který následuje uzavřený ve složených závorkách, jak je zobrazeno na obr. 26.

```
int $opakovani = 10
for ($opakovani)
{
        ;toto se provede 10 krát
}
```

Obr. 26 — Syntaxe jazyka: cyklus „for“

2.3.6. Skoky

Skoky dovolují programátorovi zavolat příkaz, který přesune vykonávání programu na požadované místo. Pozice, na kterou může kód přeskočit je návěští a je definováno pomocí příkazu „label“, za kterým je dvojtečka a vlastní pojmenování. Pomocí zavolání příkazu „goto“, dojde k přeskočení na požadované návěští. Syntaxe skoku je „goto“, symbol rovná se a jméno návěští, na které se má přeskočit. Příklad je zobrazen na obr. 27.

```
;provede se
goto(nazev)
;neprovede se
;dojde k přeskočení této části
;a pokračuje se za "label:nazev"
label:nazev
;provede se
```

Obr. 27 — Syntaxe jazyka: skoky

2.3.7. Příkazy

Pomocí příkazu „doit“ provádí aplikace většinu kontrolních operací. Syntaxe je definovaná následovně, klíčové slovo „doit“, symbol rovná se, název operace, která se má provést a volitelně za hvězdičkou data pro operaci.

Příkazem „meas“ se spouští měření konkrétním modulem aplikace. Příkaz se zapisuje klíčovým slovem „meas“, symbolem rovná se a jménem modulu, který má provést měření.

Příkaz „wait“ provede čekání po definovanou dobu. Doba čekání se definuje v milisekundách za klíčovým slovem „wait“ a symbolem rovná se.

3. Koncepční návrh aplikace na základě posouzení vstupních a výstupních parametrů úlohy

Vodopádový model je jedním ze softwarových procesních modelů. Je to nejjednodušší systematický model. Byl zvolen právě pro jeho jednoduchost. Protože není nutná kooperace týmu, nejsou schůzky se zákazníky, a jelikož jsou požadavky na systém dobře definovány na začátku procesu. Skládá se ze čtyř navazujících částí:

1. Definice a analýza požadavků
2. Návrh software
3. Implementace software
4. Testování a údržba

V této práci jsou zohledněny pouze první tři části modelu, protože testování a údržba jsou dlouhodobý proces, který bude probíhat po integraci aplikace do automatického testeru.

3.1. Definice požadavků

Požadavek je podmínka nebo schopnost, kterou musí systém splnit. Existují dva typy požadavků, a to funkční a nefunkční požadavky.

Hlavním požadavkem této práce je vytvoření interaktivní aplikace pro práci s testovacími sekvencemi s možností rozšíření na jakýkoli automatizovaný měřící systém.

3.1.1. Funkční požadavky

Funkční požadavky specifikují, co má systém dělat a jak se bude chovat vůči okolnímu světu. Otázky pro specifikaci funkčních požadavků jsou „Co má systém dělat?“, „Co má systém umět?“, „Jak má systém reagovat?“.

Funkční požadavky na interaktivní aplikaci pro práci s testovacími sekvencemi jsou následující:

- Otevřít více souborů – přepínat mezi soubory
- Editace souborů – psaní textu, kopírování, odstraňování, přesouvání objektů
- Práce se soubory – otevření, zavření, uložení
- Graficky zobrazit příkazy
- Hlídat syntaxi
- Hlídat zanoření

3.1.2. Nefunkční požadavky

Nefunkční požadavky popisují kvantifikovatelné vlastnosti systému. Otázka pro specifikaci nefunkčního požadavku je „Jaký má systém být?“.

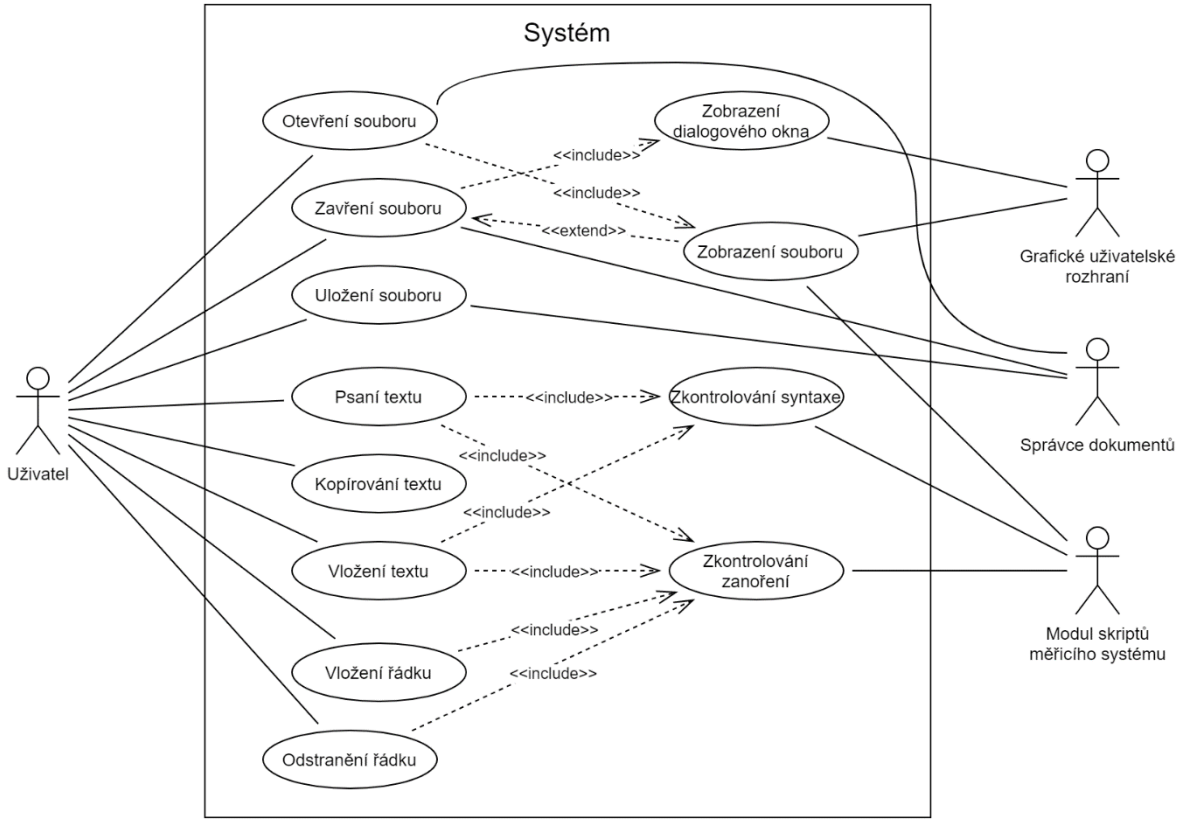
Nefunkční požadavky na tuto aplikaci jsou:

- Rozšiřitelná
- Interaktivní
- Flexibilní
- Vhodná pro dotykové obrazovky

- Integrovatelná do automatizovaného měřicího systému

3.1.3. Analýza požadavků

Funkční požadavky na aplikaci lze zobrazit do use case diagramu, viz obr. 28. Požadavky pro zobrazení souborů a zobrazení dialogového okna jsou předávány grafickému uživatelskému rozhraní. Požadavky pro práci se soubory jako otevření, zavření a uložení souboru jsou předávány správci dokumentů. Požadavky pro editaci souborů včetně zkontrolování syntaxe a zkontrolování zanoření jsou předávány modulu skriptů automatizovaného měřícího systému.



Obr. 28 — Návrh: Use Case diagram

Jelikož je vznesen požadavek na fungování s dotykovými obrazovkami, je nutné, aby měl uživatel možnost provést akci bez nutnosti použít myš, nabízí se zobrazit uživateli interaktivní menu, ve kterém bude přístup ke všem operacím, které jsou k dispozici.

Automatizovaný měřicí systém ATEster, pro který je tato aplikace navrhována je napsán v programovacím jazyce LabVIEW. Aby do něj bylo možné interaktivní vývojové prostředí integrovat, bude napsáno v LabVIEW.

3.2. Návrh aplikace

Pro snazší práci se systémem jsem se rozhodl rozdělit logiku aplikace na tři vrstvy. Každá z nich je zodpovědná za jinou úroveň operací. Jednotlivé úrovně jsou:

1. Vrstva systému
2. Vrstva modulů
3. Vrstva operací

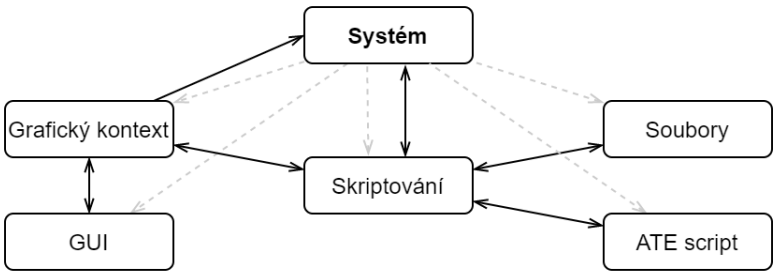
Jednotlivé vrstvy mají následující návaznost: Systémová vrstva je spuštěna, načte jednotlivé moduly a spustí je. Vytvoří sdílenou paměť a poskytne ji všem modulům, také vytvoří sdílené komunikační médium, které opět předá všem modulům. Jednotlivé moduly jsou spuštěny a zodpovědnost je předána druhé vrstvě. Každý z modulů načte svou konfiguraci a uloží jí pod svým jménem do sdílené paměti. Po načtení konfigurace se moduly inicializují a provedou nezbytné operace pro další fungování. V případě grafického uživatelského rozhraní se například zobrazí panel uživateli. V tomto stavu je aplikace spuštěna a zodpovědnost přechází na třetí vrstvu, která má za úkol reagovat na uživatelské události a provádět operace.

3.2.1. Vrstva systému

První vrstva je zodpovědná za funkce systému, spouštění, ukončování aplikace a spouštění jednotlivých modulů. Dále je vrstva zodpovědná za komunikaci mezi jednotlivými moduly. Do této vrstvy se řadí sdílené datové uložiště aplikace.

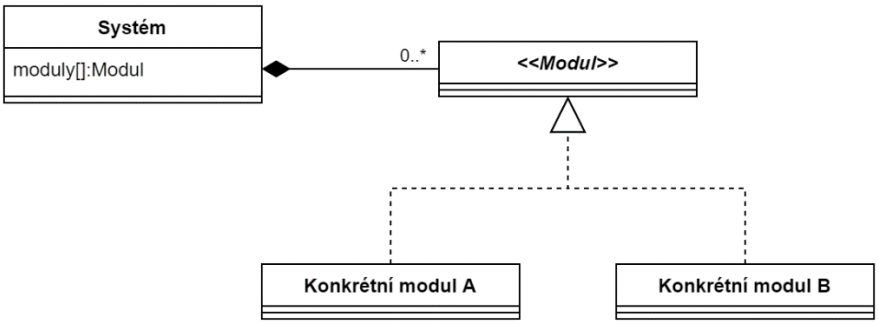
3.2.1.1. Struktura systému

Pro strukturu systému jsem zvolil tzv. plugin architekturu, kde řídícím modulem je Systém, viz obr. 29 a ten spouští všechny ostatní moduly.



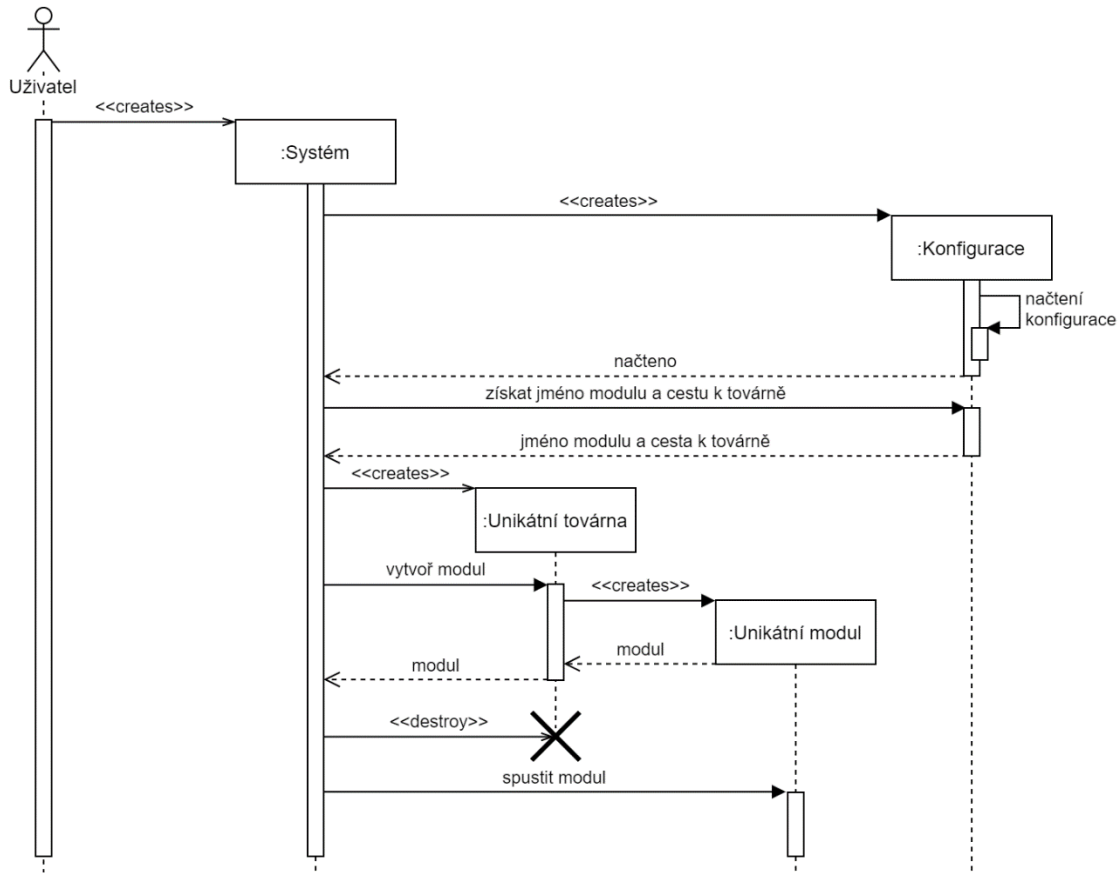
Obr. 29 — Návrh: struktura systému

Jednotlivé moduly jsou spouštěny asynchronně a paralelně. Aby bylo dosaženo větší flexibility, tak jsou moduly přiřazovány Systému z konfigurace. Systém tedy má ke každému modulu cestu a načítá jej do paměti dynamicky. Pro dosažení dynamického načítání musí mít všechny moduly jednoho společného předka, pomocí kterého budou spuštěny, jak je zobrazeno na obr. 30. Systém má tedy polymorfní datovou strukturu „moduly“, ve které jsou uloženy reference na všechny používané moduly.



Obr. 30 — Návrh: hierarchie modulu

Každý modul pracuje s jinými periferiemi a stará se o jiné operace, proto bude potřebovat pokaždé jiné závislosti. To je důvod, proč jsem zvolil návrhový vzor Abstract Factory. Každý unikátní modul je z hierarchie „Modul“. Každá unikátní továrna je z hierarchie „AbstractFactory“. Všechny závislosti jsou načítány z konfigurace. Na obr. 31 je zobrazen sekvenční diagram spouštění modulu. Na začátku uživatel spustí Systém. Systém vytvoří konfiguraci, která se načte, a vyčte z konfigurace jméno a cestu k továrně. Unikátní továrna má metodu pro vytvoření modulu, vytvoří jej a vrátí ho přes společný interface „Modul“. Továrna již nemá žádnou další funkci, a proto je ukončena. Vytvořený modul je poté asynchronně spuštěn a uchováván v polymorfní datové struktuře Systému.



Obr. 31 — Návrh: spouštění modulu

Rozšíření aplikace lze provést vytvořením nového modulu. Systému se do konfiguračního souboru poté pouze vloží jméno a cesta k novému modulu a Systém už jej samostatně spustí.

Při spuštění aplikace se spustí řídící modul Systém, který dále spouští všechny ostatní moduly a předává jim závislosti. Systém je také zodpovědný za řešení událostí, které nastanou. Při ukončení aplikace Systém požádá všechny moduly o ukončení, a nakonec ukončí i sebe. Pokud v aplikaci nastane chyba, Systém násilně ukončí všechny moduly a zobrazí uživateli chybovou hlášku.

3.2.2. Vrstva modulů

Druhá vrstva slouží k uzavření funkcionality jednotlivých modulů. Každý z nich má své vlastní zodpovědnosti a ty bude plnit. Patří zde GUI, správce dokumentů, modul automatizovaného měřicího systému a další.

Každý z modulů načítá svou vlastní konfiguraci a ukládá jí do společné paměti do své složky. Komunikovat může s ostatními moduly i se systémem pomocí společného komunikačního média.

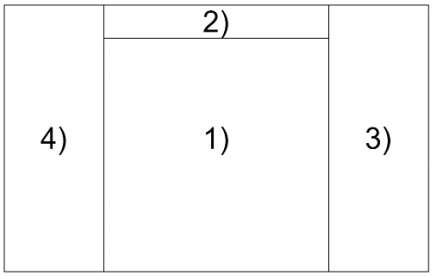
3.2.2.1. Grafické uživatelské rozhraní

Systém souží pouze pro spouštění ostatních modulů, a proto je nutné vytvořit grafické uživatelské rozhraní jako samostatný modul.

Zavedeným standardem ve vývojových prostředích je zobrazit uživateli projekt, nebo otevřenou složku na levé straně. Jelikož má být aplikace interaktivní a má být ovládána na dotykové obrazovce je nutné, aby byly jednotlivé operace proveditelné z dotykové obrazovky. Samotné operace jsou zobrazeny v menu na pravé straně. Možnost otevření více souborů je řešena pomocí záložek, které jsou zobrazeny nahoře nad pracovní plochou.

Na obr. 32 je zobrazen návrh rozložení grafického uživatelského rozhraní:

1) Pracovní plocha aplikace
2) Záložky se soubory
3) Interaktivní menu
4) Složka projektu



Obr. 32 — Návrh: rozložení GUI

3.2.2.2. Správce dokumentů

Správce dokumentů je modul, který má na starosti operace se soubory. Modul spouští dialogové okno pro otevření souboru. Také má informaci o tom, které soubory jsou již otevřeny a jejich uložený nebo neuložený stav. Tento modul dále spravuje informaci o tom, který ze souborů je právě otevřen a editován. Další zodpovědností tohoto modulu je spravovat domovskou obrazovku.

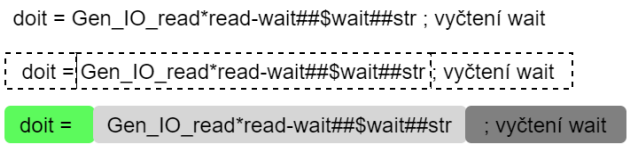
Modul má přímý přístup k záložce se soubory z grafického uživatelského rozhraní a zobrazuje na ni jednotlivé otevřené soubory.

3.2.2.3. Modul automatizovaného měřicího systému

Data k otevřeným souborům a všechny operace, které se souborů týkají jsou zodpovědností tohoto modulu. Mezi jednotlivé operace patří zápis textu, kontrola syntaxe, hlídání zanoření, načítání souborů, ukládání souborů a další.

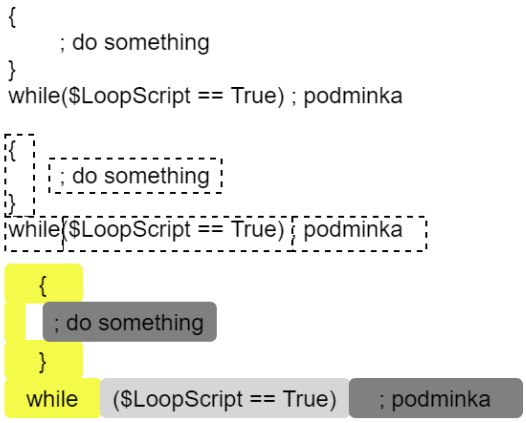
V tomto modulu jsou izolovány všechny závislosti, které se týkají operací se skripty ATEsteru. Všechny operace jako odstranění řádku, zapsání textu, vygenerování grafických objektů se dějí zde. Tato izolace závislostí je velmi důležitá pro pozdější možnost rozšíření. Když budou všechny závislosti izolovány neexistuje problém v tom vytvořit modul pro práci s úplně jiným druhem souborů a závislosti potřebné pro něj budou izolovány v něm.

Ze syntaxe jazyka popsané v samostatné kapitole 2.3 je patrné, že skripty obsahují dva základní typy příkazů, a to příkazy jednořádkové a víceřádkové bloky příkazů. Jedním z požadavků a cílů práce je grafické zobrazení těchto skriptů. Na obr. 33 je zobrazen návrh grafického zobrazení samostatného příkazu. Samostatný příkaz lze rozdělit na tři části, a to typ příkazu, data příkazu a komentář. Typ příkazu je podbarven barevně, aby bylo na první pohled zřetelné, o jaký příkaz se jedná a aby hledání typu příkazu mezi ostatními bylo výrazně urychleno. Díky podbarvení typu příkazu lze také hlídat syntaxi, protože pokud příkaz nebude rozeznán, tak nedojde k jeho podbarvení, což indikuje chybu syntaxe. Text příkazu je pro každý řádek specifický, a proto je označen neutrální barvou. Komentář je zobrazen tmavě šedou, aby nebral pozornost od hlavního kódu a aby bylo možné jej opticky filtrovat.



Obr. 33 — Návrh: samostatný příkaz

Blok příkazů lze rozdělit na čtyři části: typ příkazu, data příkazu a komentář, což je stejné jako pro obyčejný příkaz, ale má navíc tělo, které vymezuje jeho působnost. Návrh bloku příkazů je zobrazen na obr. 34. Uvnitř bloku příkazů mohou být samostatné příkazy nebo další bloky příkazů. Pro grafické ohraničení bloku jsou indentační symboly podbarveny stejnou barvou jako příkaz, kterému patří. Mezi indentačními symboly je spojnice, která vymezuje oblast působení.



Obr. 34 — Návrh: blok příkazů

Jelikož je podbarvení prováděno na základě typu příkazu je nutné jej rozeznat z čistého textu, rozhodl jsem se využít pro definování syntaxe skriptovacího jazyka regulární výrazy. Regulérní výraz, nebo také regex je speciální textový řetězec pro popis a vyhledávání vzorů v textu[8].Tyto regulární výrazy jsou definovány v konfiguraci a je možné tímto syntaxi kdykoli změnit nebo upravit bez nutnosti sestavovat aplikaci znovu.

3.2.2.4. Modul skriptů

Pro dosažení co největší flexibility je chování systému řešeno pomocí skriptů. Modul skriptů se stará o jejich načítání spouštění, ale také vykonávání.

3.2.3. Vrstva operací

Třetí vrstva slouží pro práci mezi moduly. Umožňuje funkcionalitu a provádění požadovaných uživatelských operací. Všechny činnosti systému jsou řešeny v této vrstvě z důvodu co největší flexibility. Tato vrstva je celá definovaná v konfiguraci aplikace a je možné ji změnit bez nutnosti znovu sestavit aplikaci. Pro ovládání této vrstvy byl vytvořen vlastní skriptovací jazyk.

3.2.3.1. Skriptování aplikace

Skriptovací jazyk slouží k ovládání veškeré funkcionality aplikace. Každá událost v systému spouští svůj odpovídající skript a ten vykoná požadovanou operaci. Ať už se jedná o stisknutí tlačítka menu, nebo zavření aplikace.

Skriptovací modul aplikace má přístup ke všem ostatním modulům a komunikuje s nimi pomocí komunikačního kanálu. Vzhledem k tomu, že komunikační kanál je asynchronní a některé operace se musí provádět sekvenčně, je nutné vytvořit synchronizační vazbu. K vytvoření vazby je mimo jiné využíváno společné datové uložiště. Při zaslání asynchronní zprávy začne cílový modul vykonávat požadovanou operaci a jakmile ji dokončí nastaví hodnotu proměnné ve sdíleném datovém uložišti na požadovanou hodnotu. Skriptovací modul odešle zprávu a čeká, dokud není proměnná nastavena na požadovanou hodnotu.

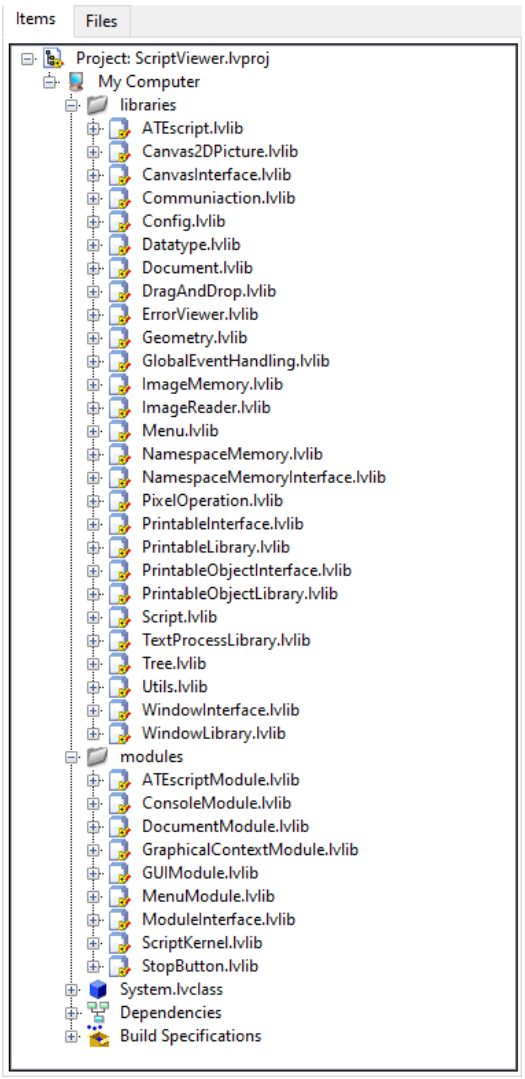
Vzhledem k tomu, že sdílené datové uložiště obsahuje pouze textové řetězce, byl vytvořen systém datových typů, který je reprezentovaný pomocí těchto textových řetězců.

4. Vytvoření interaktivní aplikace

Cílem této části práce je popis základních částí vytvořené interaktivní aplikace pro práci s testovacími sekvencemi. Aplikace je napsaná v LabVIEW 2020 64 bit z důvodu pozdější integrace do automatizovaného testeru ATEster, který je napsaný v této verzi LabVIEW.

4.1. Struktura projektu

Při tvorbě projektu bylo myšleno na maximální znovu-použitelnost jednotlivých tříd, z toho důvodu je každá samostatná část uložena v knihovně s co nejmenším počtem závislostí. Projekt obsahuje dvě složky, a to složku s knihovnami a složku s moduly, jak je zobrazeno na obr. 35. Každá knihovna obsahuje třídy, které zajišťují společný zájem.

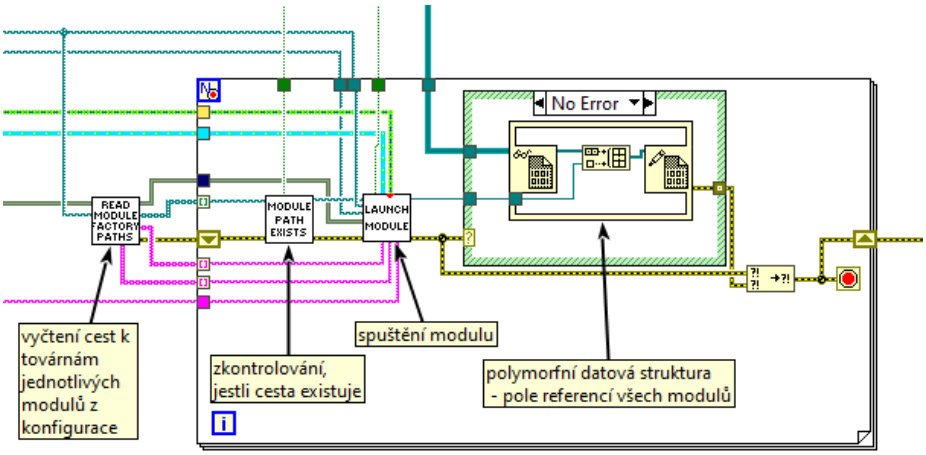


Obr. 35 — Implementace: projekt

Ústředním prvkem celé aplikace je třída „System“, která obsahuje metodu main.vi. Jedná se o spouštěcí VI aplikace. Při spuštění vytvoří společné komunikační médium a sdílenou datovou paměť aplikace. Poté načte svou konfiguraci a spustí ostatní moduly. Po spuštění modulů slouží jako prostředník pro řešení systémových událostí a chyb.

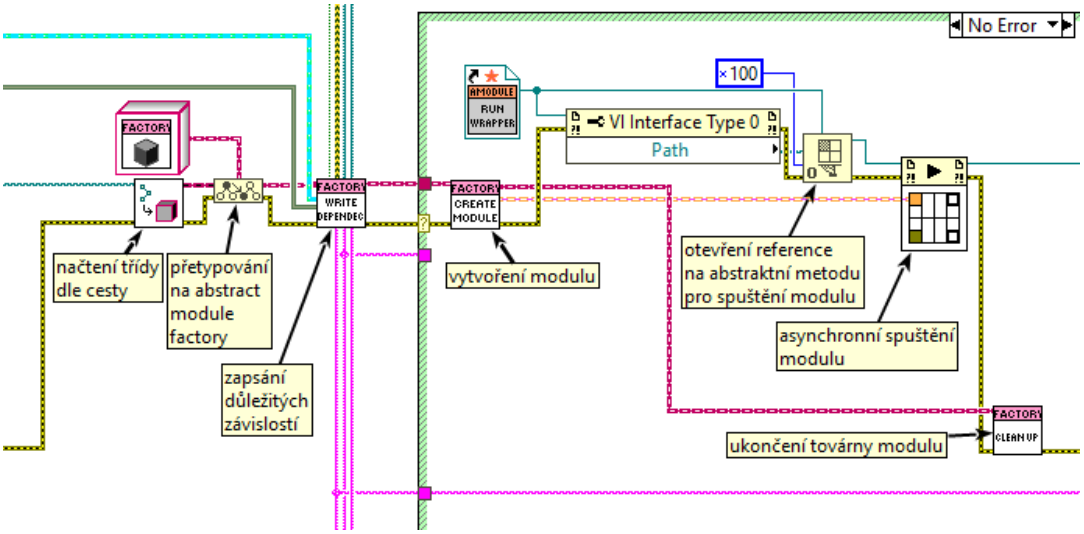
4.1.1. Spouštění modulů

Moduly jsou spouštěny na základě jejich jména a cesty k jejich továrně v hierarchii projektu. Systém nejprve načte cesty ke všem modulům, které má spustit, jak je zobrazeno na obr. 36. Po načtení cest zkontroluje, jestli jsou cesty v pořádku, asynchronně spustí jednotlivé moduly a reference na moduly uloží do polymorfní datové struktury pro možnost pozdějšího ukončení modulů.



Obr. 36 — Implementace: inicializace systému

Samotné spuštění modulu probíhá následovně, viz obr. 37. Nejprve se dynamicky načte třída továrny modulu. Při načtení třídy je třída typu „LV Object“, proto je nutné ji přetypovat na třídu „AbstractModuleFactory“. Po přetypování se do továrny zapíšou všechny závislosti, které systém modulům poskytuje. Továrna si uloží své potřebné závislosti a vytvoří modul, pro který je určena. Otevře se reference pro reentrantní spouštění a asynchronně se modul spustí. Po spuštění modulu je otevřená reference předána zpět systému pro uložení a továrna modulu je ukončena.



Obr. 37 — Implementace: asynchronní spuštění modulu

4.1.2. Architektura modulu

Každý modul je vytvořen jako frontový stavový stroj za použití producent-konzument architektury. Producentem pro moduly jsou systém, skriptovací modul, ale také jiné moduly. Grafické uživatelské

rozhraní má jako producenta také uživatelskou smyčku s Event strukturou pro zachytávání uživatelských operací, protože se jedná o jediný modul, který interaguje s uživatelem.

Jakmile je modul spuštěn, načte vlastní konfiguraci, provede svou inicializaci a nahlásí systému, že je připraven pracovat. Jakmile všechny moduly nahlásí, že jsou spuštěny, je spuštění aplikace dokončeno. Od té doby moduly pracují paralelně jako samostatné jednotky a komunikují spolu pomocí společného komunikačního média.

4.1.3. Komunikace mezi moduly

Komunikace mezi moduly je zajištěna pomocí dvou tříd: „LocalCommunication“ pro lokální komunikaci v modulu a „GlobalCommunication“ pro globální komunikaci mezi moduly. Každý modul má vlastní lokální komunikaci a jen skrze ni může zkontaktovat jiné moduly. Třída pro lokální komunikaci obsahuje frontu pro naplnění funkce stavového stroje. Tato fronta může přijímat příkazy z vlastní třídy lokální komunikace nebo ze třídy globální komunikace. Při inicializaci lokální komunikace se automaticky reference na frontu předává globální komunikaci s volacím jménem. Při požadavku na odeslání zprávy třída lokální komunikace rozpozná, jestli je požadavek lokální nebo globální podle vstupního parametru adresáta. Pokud požadavek na odeslání zprávy není pro lokální frontu, je požadavek předán globální komunikaci, ta provede vyhledání příslušného adresáta mezi zapsanými frontami a pokud nalezne shodu, zapíše požadavek do cílové fronty.

4.2. Grafický systém

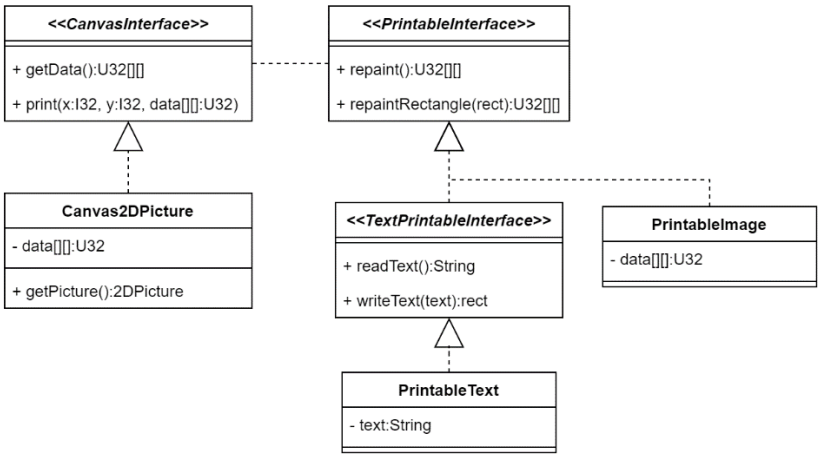
Pro zobrazení jednotlivých příkazů jako objekty bylo nutné vytvořit grafický systém. Použití klasických prvků LabVIEW neposkytnulo dostatečnou flexibilitu pro vytvoření navrženého grafického vzhledu příkazů. LabVIEW obsahuje plátno „2D Picture“, na které lze tisknout některé základní grafické obrazce jako kružnice, čtverec a další. Hlavní funkce, které bylo využito, je možnost vytisknutí 2D bitmapy. Pomocí tisku 2D pole pixelů lze dosáhnout plné flexibility a lze vytvořit jakýkoli grafický obrazec.

Logika práce s grafickým plátnem je rozdělena na tři úrovně: úroveň tisknutelných dat, úroveň tisknutelných objektů, úroveň tisknutelných oken.

4.2.1. Tisknutelná data

Nejnižší úroveň pohledu na tisknutelná data je dvou dimenzionální pole pixelů, které se tisknou na plátno. Toto 2D pole je uchováno ve třídě „Canvas2DPicture“. Pro vytisknutí obrázku na pozici do tohoto plátna existuje v této třídě metoda pro vytisknutí dat „print()“. LabVIEW nepodporuje práci s transparentností při tisku. Pro umožnění větší flexibility pro uživatele a lepší možnost znovu-použití této části systému byl implementován do pixelů alfa kanál. Jednotlivé pixely jsou uloženy jako neznaménkové třiceti-dvou bitové celé číslo. Toto číslo je rozděleno na čtyři části po osmi bitech a je reprezentováno jako červená, zelená, modrá a alfa. Alfa kanál je využíván při tisknutí dvou bitmap přes sebe. Čím vyšší hodnotu alfa kanál nového pixelu má, tím více bude vrchní pixel sytý. Při nulové hodnotě alfa kanálu se jedná o zcela transparentní pixel.

Základní objekty pro tisk jsou obrázek a text. Pomocí nich lze vytvořit jakýkoli grafický prvek s dostatečnou flexibilitou. Hierarchie tříd pro tisknutelná data je zobrazena na obr. 38. Nejnižší tisknutelný prvek hierarchie je „PrintableInterface“. Jelikož je tato aplikace v jádře textovým editorem, existuje také interface „TextPrintableInterface“, který slouží jako základní tisknutelný text. Všechny textové objekty implementují tento interface.

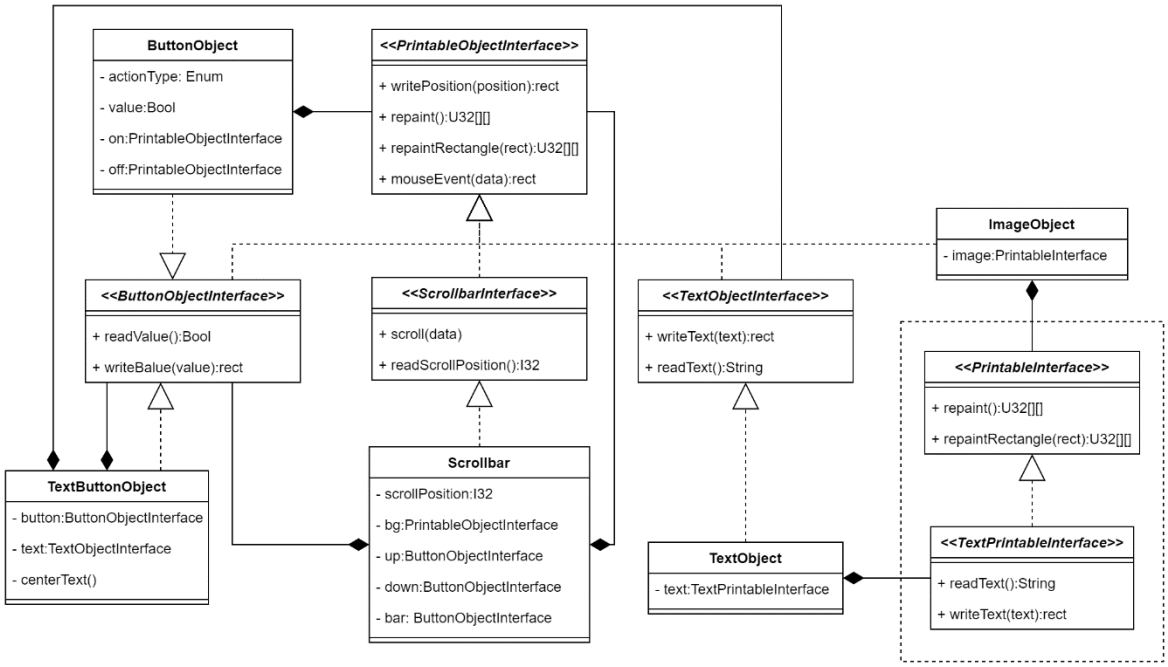


Obr. 38 — Implementace: hierarchie tisknutelných dat

4.2.2. Tisknutelné objekty

Vytisknutí jednotlivých grafických prvků pro dostatečnou práci se skripty nestačí, a proto existuje vrstva tisknutelných objektů. Tato vrstva kombinuje základní tisknutelné prvky a přidává k nim chování. Tisknutelné objekty dokážou reagovat na uživatelské události a měnit svůj vzhled pomocí změny pozice a přepínání zobrazených tisknutelných objektů. Všechny tisknutelné objekty jsou komponovány ze tříd tisknutelných dat.

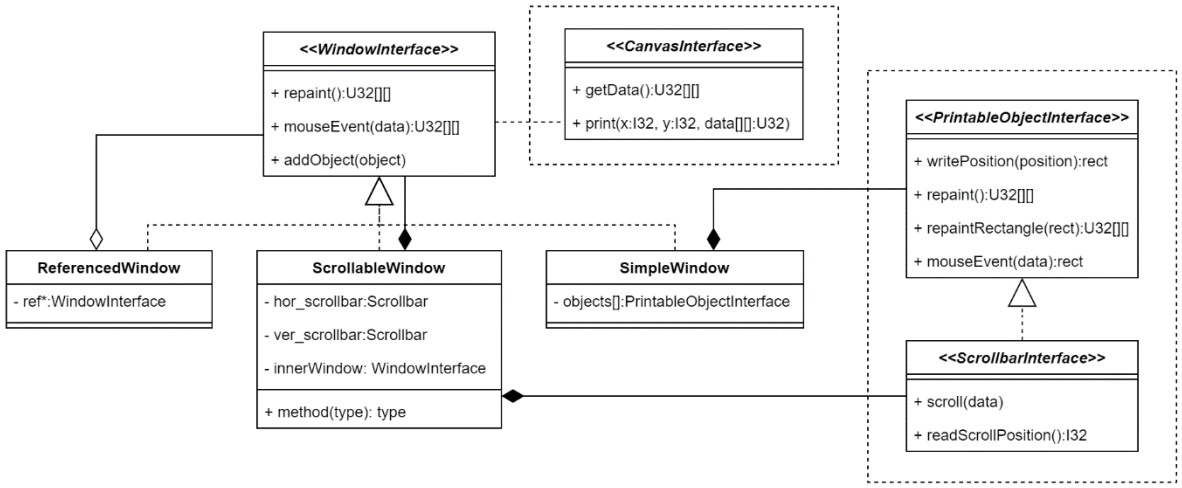
Hierarchie tisknutelných objektů je zobrazena na obr. 39. Všechny tisknutelné objekty musí implementovat interface „PrintableObjectInterface“, který všem dává povinnost umět vytisknout svá data, změnit pozici a reagovat na uživatelské události. Z tohoto interface jsou odvozeny další interface pro zachycení specifické funkcionality jednotlivých objektů, jako například tlačítko, posuvník, obrázek.



Obr. 39 — Implementace: hierarchie tisknutelných objektů

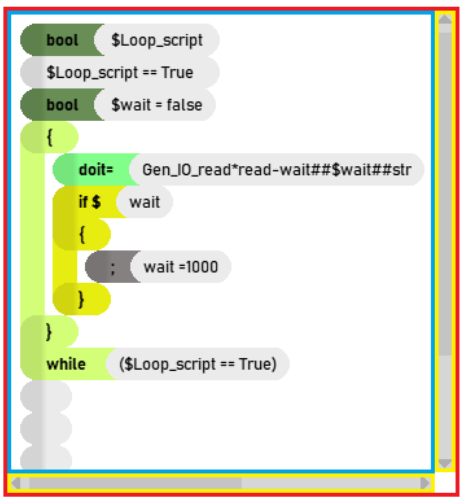
4.2.3. Tisknutelná okna

Vzhledem k tomu, že jedním z požadavků je možnost otevření více skriptů a přepínání mezi nimi, nebylo by vhodné pokaždé vytvářet novou objektovou strukturu při každém přepnutí souboru. To je jedním z důvodu, proč existuje vrstva tisknutelných oken. Jedná se o vrstvu, která umožňuje sdružovat tisknutelné objekty v izolované formě. Hierarchie tisknutelných oken je zobrazena na obr. 40. Tisknutelná okna jsou komponována ze tříd tisknutelných objektů.



Obr. 40 — Implementace: hierarchie tisknutelných oken

Díky této vrstvě bylo možné vytvořit posouvatelné okno, které pokaždé zobrazuje pouze část vnitřního okna, která se vleze na obrazovku. Skripty jsou často delší, než je velikost obrazovky a pomocí posouvatelného okna je možné odebrat starost o posouvání z modulu automatického testeru a předat ji na jiné místo. Pro modul automatizovaného testeru je posouvání zcela transparentní a má k dispozici teoreticky neomezeně velké okno pro tisk svých objektů. Uživatel může posouvat skripty pomocí posuvníků. Na obr. 41 je zobrazena struktura vnoření oken. Červený rámeček ohraničuje vnější posouvatelné okno, které má v sobě tři prvky, a to horizontální a vertikální posuvníky, které jsou označeny žlutou barvou a vnitřní okno, které je označené modrou barvou. Vnitřní okno může mít libovolnou velikost a je zobrazena pouze část, která se vleze na obrazovku. Uživatel pomocí posuvníků posouvá zobrazovanou podmnožinu vnitřního okna.



Obr. 41 — Implementace: vnoření oken

4.3. Systém zobrazení příkazů

Zobrazování jednotlivých příkazů je specifické pro modul automatického testeru. Při rozšíření aplikace na práci s jinými typy souborů by objekty byly zobrazovány jiným modulem. Byl zvolen tří–úrovňový přístup s textovou reprezentací, reprezentací struktury a grafickou reprezentací jednotlivých příkazů.

4.3.1. Textová reprezentace

Na nejnižší úrovni jsou soubory reprezentovány jako textové řetězce. Jedná se o stejnou reprezentaci jako při textovém zobrazení. V této úrovni probíhá interakce uživatele se souborem. Odstranění řádku, přidání řádku, zapsání textu a vyčtení textu jsou základní operace, pomocí kterých lze provést kteroukoli další operaci. Operace pro vrácení a zopakování akcí jsou implementovány v této úrovni.

4.3.2. Reprezentace struktury

Vzhledem k tomu, že příkazy mají být reprezentovány jako grafické objekty, bylo nutné vytvořit vrstvu struktury skriptu, ve které jsou příkazy reprezentovány pomocí jejich popisných informací. Ke každému příkazu je skladována informace o typu příkazu, typ příkazu, text příkazu, text komentáře, úroveň zanoření a informace, jestli se jedná o rodičovský příkaz bloku příkazů. Z těchto informací je možné vygenerovat grafické zobrazení příkazů.

4.3.3. Grafická reprezentace

Z informací reprezentace struktury je možné vygenerovat grafické zobrazení příkazu. Každý typ příkazu může mít unikátní grafickou reprezentaci všech jeho částí. Samostatné části jsou načítány ze souboru z konfigurace. Dle typu příkazu se vybere typ tisknutelných objektů, které jsou vytvořeny. Třetí vrstva systému zobrazení příkazů ukládá a vytváří tyto tisknutelné objekty.

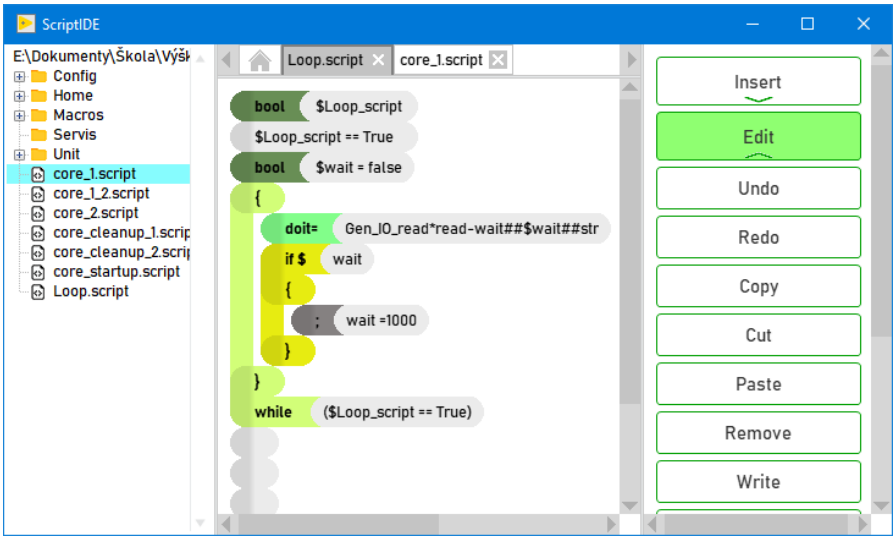
4.3.4. Kooperace vrstev

Vazba mezi těmito vrstvami je realizována pomocí čísla řádku. Každý příkaz tedy existuje v textové, strukturální a grafické reprezentaci. Každá z těchto vrstev má specifický význam. Textová vrstva slouží pro práci se skriptem na úrovni editace souboru. Strukturální vrstva slouží jako výchozí bod pro grafickou reprezentaci a pro strukturování celého souboru. Grafická vrstva slouží pro vytvoření tisknutelných objektů, zobrazení příkazů uživateli a pro zachycení uživatelských požadavků.

Příklad funkce této architektury je vidět při požadavku uživatele o zakomentování příkazu. V grafickém uživatelském rozhraní uživatel klikne na příkaz. Grafická vrstva zpracuje událost, protože zde jsou příkazy reprezentovány jako objekty a je porovnatelná pozice kliknutí a pozice příkazu. Druhým krokem je zjištění, jestli se již jedná o komentář nebo o jiný typ příkazu. Tuto informaci systém získá ze strukturální reprezentace souboru. Na základě získané informace se vyčte z textové reprezentace text příkazu a upraví se takovým způsobem, aby bylo dosaženo zakomentování. V případě skriptovacího jazyka pro ATEster se jedná o přidání středníku před text příkazu. Takto upravený text je zpětně zapsán do textové reprezentace. Protože se změnou textové reprezentace se mění také strukturální a grafická reprezentace je pro daný příkaz vygenerována nová strukturální reprezentace a na základě té je upravena grafická reprezentace a ta je vytištěna uživateli.

4.4. Grafické uživatelské rozhraní

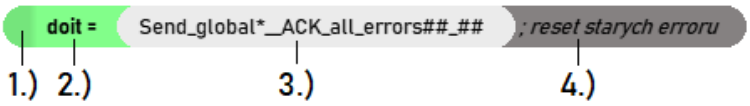
Dle návrhu grafického uživatelského rozhraní byl vytvořen modul grafického uživatelského rozhraní. Pracovní plocha, interaktivní menu a záložky s otevřenými soubory jsou reprezentovány pomocí prvku „2D Picture“. Všechny grafické prvky jsou vytvořeny pomocí grafického systému, který byl popsán výše v kapitole 4.2. Složka projektu je reprezentována pomocí knihovního prvku LabVIEW „Tree“. Vzhled grafického uživatelského rozhraní je zobrazen na obr. 42.



Obr. 42 — Implementace: GUI

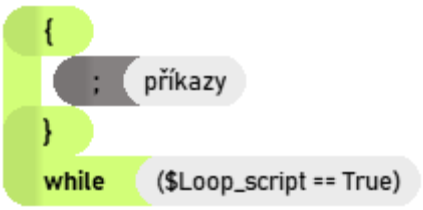
4.4.1. Příkazy

Jednotlivé příkazy skriptů jsou zobrazovány na čtyři části. Na obr. 43 je zobrazen jeden příkladový příkaz. Poznámka 1.) je uchopovací bod příkazu. Pro přesunutí příkazu na jinou pozici je nutné příkaz uchopit za tento bod. 2.) Typ příkazu je podbarvený, aby byly jednotlivé druhy příkazů jasně odlišitelné na první pohled a aby se usnadnilo hledání specifického typu příkazu ve velkých skriptech. 3.) Text příkazu je podbarven neutrální šedou barvou, protože je pro každý příkaz jiný. 4.) Komentář příkazu je podbarven šedou barvou, aby byl opticky potlačený od ostatního textu a nebral pozornost uživatele.



Obr. 43 — Implementace: příkaz

Bloky příkazů korespondují s návrhem a jsou barevně sladěny s rodičovským příkazem bloku. V případě obr. 44 jsou indentační znaky podbarveny stejnou barvou jako příkaz „while“. V případě, že by nebylo dodrženo správné závorkování indentačními symboly, byly by špatně rozpoznané symboly podbarveny červenou barvou.



Obr. 44 — Implementace: blok příkazů

4.4.2. Menu

Menu je tvořeno jako hierarchická stromová struktura. Obsahuje dva druhy tlačítek, ve stromové struktuře se jedná o listy a větve, kde listy jsou tlačítka pro konkrétní operace a větve jsou tlačítka pro zobrazení, nebo skrytí podstromu. Menu se dynamicky mění na základě kontextu aplikace. Je toho dosaženo tím, že se pro každý kontext zobrazí konkrétní okno menu. Pro domovskou obrazovku nedávají smysl operace editaci souborů, a proto nejsou uživateli zobrazeny. Při otevření souboru se menu automaticky změní a zobrazí všechny relevantní operace.

4.4.3. Záložky

Panel záložek slouží ke zobrazení otevřených souborů a k přepínání mezi nimi. Uživatel může přejít na domovskou obrazovku aplikace otevřením záložky se symbolem domu. Mezi jednotlivými záložkami lze přepínat pomocí kliknutí na požadovanou záložku. Že je záložka otevřena je indikováno podbarvením šedou barvou. Na obr. 45 je zobrazen panel záložek se dvěma otevřenými soubory. Každá záložka obsahuje indikátor neuloženého stavu souboru. Při editaci souboru se nad záložkou rozsvítí oranžový proužek, který indikuje, že soubor je editován a pro uložení změn do souboru je nutné provést operaci uložení. Každou záložku je možné zavřít pomocí tlačítka se symbolem křížku.

Obr. 45 — Implementace: záložky

4.4.4. Grafický kontext

Grafické uživatelské rozhraní samo o sobě neobsahuje objekty pro tisknutí, ale je pouze prostředníkem pro zobrazování dat uživateli. Obsahuje tři plátna, na které je možné kreslit. Požadavek pro vykreslení dat je možné zaslat na konkrétní plátno pomocí komunikačního média systému.

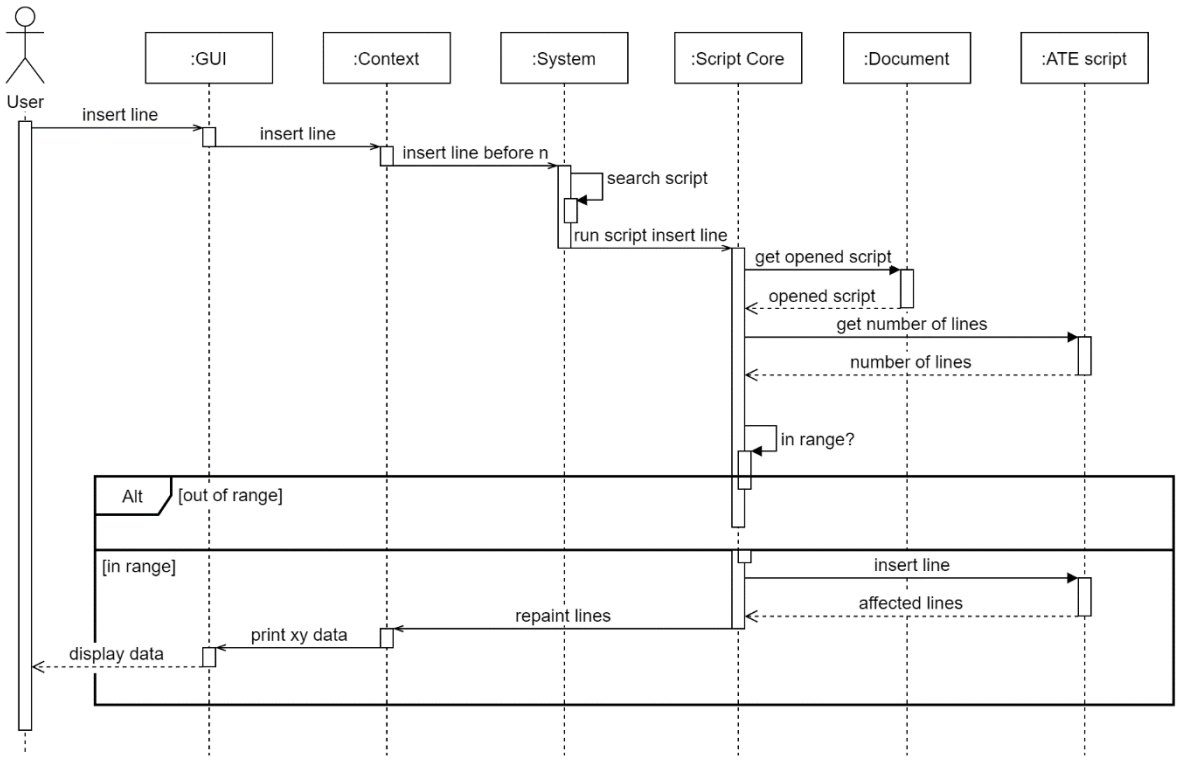
Grafický kontext je modul, který slouží pro práci s objekty. V něm jsou uloženy jednotlivá tisknutelná okna, která obsahují tisknutelné objekty. Všechny události, které v grafickém uživatelském rozhraní

vyvstanou jsou pouze zaznamenány a předány grafickému kontextu daného plátna. Pro obsluhu tří pláten jsou spuštěny tři grafické kontexty.

4.5. Zpracování události

Jednotlivé události jsou zpracovány skriptovacím modulem. Pro aplikaci byl vytvořen vlastní skriptovací jazyk. Tento skriptovací jazyk je napojen na komunikační médium systému a může komunikovat se všemi moduly. Jako datové uložiště pro proměnné využívá sdílené datové uložiště aplikace. Při spuštění skriptu se vytváří složka ve sdíleném datovém uložišti pro lokální proměnné. Lokální proměnné tedy mají platnost pro daný skript a je možné je měnit v kterékoli části skriptu. Po ukončení skriptu se složka lokálních proměnných ukončuje. Skript může provádět operace na celé datové paměti, příkladem takových operací je sčítání, odčítání, násobení, zapsání na index a další. Skript může zavolat operaci na kterémkoli modulu a tím řídí chování aplikace. Skript může také spustit vnořený skript.

Na obr. 46 je zobrazen příklad zpracování uživatelského požadavku na vložení řádku. Uživatel interaguje přímo s GUI a vytvoří požadavek na vložení řádku poklepáním na plátno. GUI pouze zobrazuje data, ale nerozumí jim, proto předává požadavek na grafický kontext odpovídajícího plátna. Grafický kontext obsahuje tisknutelné okno s tisknutelnými objekty a vyhledá ten, pro který odpovídají souřadnice požadavku. Systém je zodpovědný za reagování na události, z toho důvodu je požadavek na vložení řádku i s informací o tom, ve kterém řádku k požadavku došlo, předán systému. Systém vyhledá skript, který odpovídá dané události a pošle požadavek na spuštění daného skriptu na modulu skriptovacího jádra. Skriptovací jádro spustí odpovídající skript a začne jej vykonávat. Nejprve zjistí z modulu správce dokumentů, který soubor je otevřený. Po zjištění se zeptá modulu automatického testeru, kolik řádků má daný soubor a vyhodnotí, jestli je požadavek v rozmezí souboru. Pokud vznikl požadavek mimo rozmezí souboru, nic se dále neprovádí, skript se ukončí a událost končí. Pokud požadavek vznikl v rozmezí, tak skript pošle požadavek na vložení řádku do aktuálně otevřeného souboru. Při vložení řádku mohou být ovlivněny některé jiné řádky. Informace o tom, které řádky byly ovlivněny je předána zpět skriptu a ten už pouze přepošle požadavek na překreslení grafickému kontextu. Grafický kontext vygeneruje vytisknutelná data a pošle je na GUI. GUI data zobrazí uživateli, a tím sekvence zpracování události končí.



Obr. 46 — Implementace: zpracování události

5. Ověření funkce a zhodnocení výsledků

Byla vytvořena interaktivní aplikace pro práci s textově orientovanými skriptovanými testovacími sekvencemi. Primárně byla aplikace navržena pro práci s testovacími sekvencemi pro automatický tester ATEster s myšlenkou pozdější integrace.

5.1. Rozšiřitelnost, znovu-použitelnost

Aplikace byla navržena modulárně s možností rozšíření na jakýkoli druh textových, ale i jiných souborů. Pro rozšíření aplikace stačí vytvořit unikátní modul pro cílový typ souborů a přidat jej systému. Jednotlivé části systému byly izolovány do samostatných knihoven pro možnost pozdějšího znovu-použití třeba ve zcela jiném kontextu.

5.2. Práce se soubory

V aplikaci je možno otevřít více souborů najednou. Všechny otevřené soubory jsou přehledně zobrazeny uživateli v panelu záložek a uživatel mezi nimi může jednoduše přepínat. Při neotevřeném žádném souboru je uživateli zobrazena domovská obrazovka s posledními otevřenými soubory, aby se mohl co nejrychleji a snadno vrátit zpět k práci, kde skončil. Aplikaci lze rozšířit pro otevření více druhů souborů najednou a všechny je zobrazit na panelu záložek. Při změně syntaxe skriptů automatického testeru není nutné vstoupit do kódu a cokoli upravovat. Veškerá syntaxe je definovaná v konfiguraci a je možné ji kdykoli snadno změnit bez nutnosti znovu sestavovat aplikaci. Tak jako syntaxe je definovaná z konfigurace, tak jsou všechny grafické prvky definovány mimo aplikaci. Toto umožňuje aktualizovat grafickou stránku aplikace s vyvíjejícími se trendy v designu software bez nutnosti zásahu do kódu.

5.3. Editace souborů

Soubory mohou být editovány stejným způsobem jako v textovém editoru, je možné zapisovat text, kopírovat řádky, odstraňovat řádky a další. Pro jasné určení, se kterým příkazem se pracuje je implementováno označování. Řádek, který je aktivní je podbarven jasnou odlišnou barvou. Při označení řádku, který je mimo obrazovku se zobrazení posune tak, aby označený objekt byl vždy viditelný. Vzhledem k tomu, že aplikace je určena do průmyslu, kde se ve velké míře používají dotykové obrazovky, jsou na příkazech uchopovací body, za které je lze přesouvat. Ze stejného důvodu je na grafickém uživatelském rozhraní zobrazeno interaktivní menu, které se dynamicky mění dle kontextu. Toto menu obsahuje všechny právě dostupné funkce. Menu je strukturované jako stromová hierarchie.

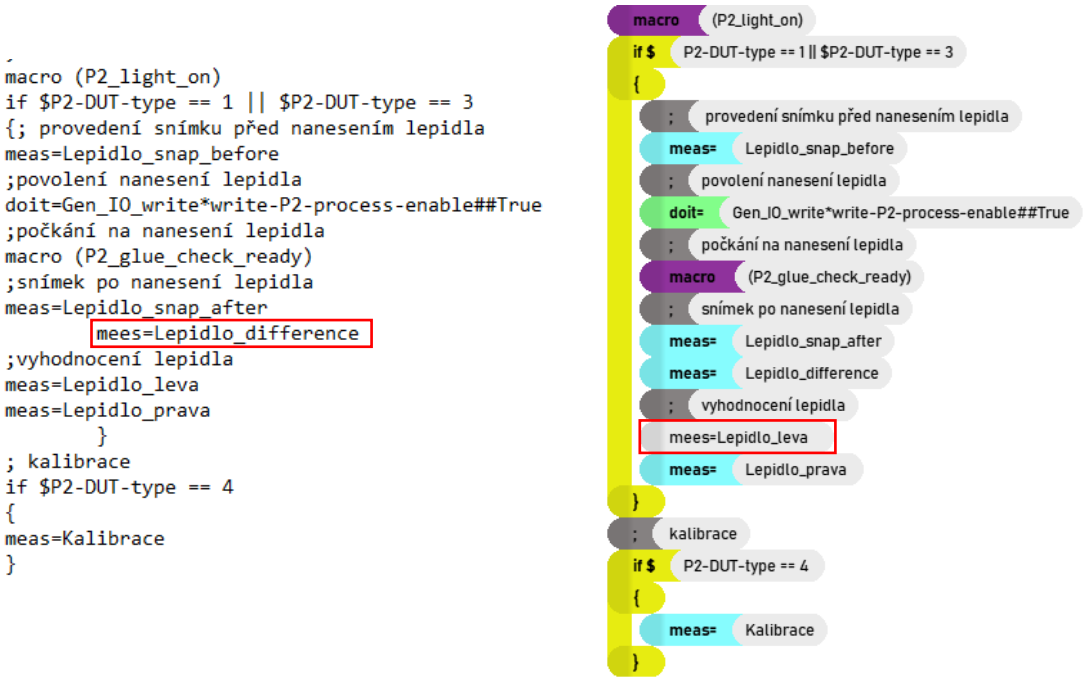
5.4. Grafické zobrazení

Klíčovou funkcí této aplikace je grafické zobrazení příkazů. Příkazy jsou zobrazeny jako celistvé objekty a jsou podbarvovány podle jejich typu. Komentáře jsou zobrazeny odlišnou barvou, aby nebraly pozornost uživatele. Bloky příkazů jsou jasně ohraničeny a odsazeny, aby bylo jasně viditelné, které příkazy do bloku spadají a které ne. Implementace byla provedena takovým způsobem, aby uživatel nebyl schopen narušit celistvost ohraničení.

5.4.1. Porovnání textového a grafického zobrazení

Textové i grafické zobrazení mají své výhody a nevýhody. Porovnání stejné testovací sekvence zobrazené v textové a grafické podobě je na obr. 47. Hlavní výhodou grafického oproti textovému zobrazení je hlídání syntaxe a podbarvení příkazů. Tato výhoda je obzvláště užitečná pro začátečníky.

Při překlepu, nebo při nepřesném zapsání syntaxe nedojde k podbarvení příkazu a je jednoznačně indikováno, že příkaz nebyl rozeznán. Takový překlep je zobrazen na obr. 47 v červeném rámečku. Na levé straně je zvýrazněný překlep v textovém zobrazení a na pravé straně je zobrazen stejný překlep v grafickém zobrazení. U grafického zobrazení nedošlo k podbarvení modrou barvou, tak jako u všech ostatních příkazů stejného typu.



Obr. 47 — Zhodnocení: porovnání textového a grafického zobrazení

Hlavní výhodou grafického zobrazení je, že testovací sekvence je vnímána více smysly. Jsou jasně ohraničené příkazy a bloky příkazů, kde je zcela jasné, které příkazy do bloku patří a které ne. S grafickými objekty je možné pracovat na dotykových obrazovkách a pro zápis textu využít systémovou grafickou klávesnici. Může být jednodušší pro začátečníky a pro lidi, kteří nejsou programátoři. Popisování barevně odlišeného kódu je snazší. V aplikaci je implementována kontrola syntaxe a při nedodržení nedojde k podbarvení příkazu, a to uživatele vede ke kontrole ještě před spuštěním. Hlídání zanořování je velmi užitečné pro komplikované struktury s mnoho vnořenými bloky. Grafické zobrazení automaticky formátuje text a všichni uživatele vidí stejně formátovaný text. Při textově orientovaném zobrazení si programátor určuje formátování sám a při předání kódu někomu jinému s jiným stylem formátování bude delší dobu trvat, než se s kódem seznámí. Při automatickém formátování je pro všechny programátory kód formátován stejně a jeden programátor se nemusí přizpůsobovat formátovacímu stylu jiného programátora.

Závěr

Byla navržena interaktivní aplikace pro práci s testovacími sekvencemi a jejich grafickou reprezentaci. Při návrhu byla zohledněna myšlenka znovu-použitelnosti, modularity a možnosti rozšíření na jakýkoli textově orientovaný typ souborů. Návrh byl podřízen tomu, aby aplikace mohla být využita lidmi, kteří nemají zkušenosti s programováním a umožnila tak automatizaci testovacích sekvencí více uživatelům.

Po provedení systematické analýzy programového prostředí LabVIEW a problému samotného, byla aplikace vytvořena pomocí objektově orientovaného programování, což umožňuje pozdější integraci do automatizovaného testeru. Všechny části aplikace jsou uzavřeny do samostatných modulů, které jsou asynchronně paralelně spouštěny. Chování aplikace je řízeno skriptovacím modulem, který interpretuje vlastní skriptovací jazyk. Aplikace podporuje otevření více souborů a jejich editaci. Jednotlivé příkazy jsou reprezentovány jako grafické objekty a je možné je přesouvat na dotykových obrazovkách. Bloky příkazů jsou výrazně ohraničeny a je jednoznačně viditelné, které příkazy do bloku patří a které ne. Je implementována kontrola syntaxe, aby bylo předcházeno syntaktickým chybám při tvorbě skriptů.

Vývoj této aplikace má za cíl přivedení více lidí ke skriptování testovacích sekvencí. Grafická reprezentace skriptovaných testovacích sekvencí má přínos pro uživatele, kteří nemají zkušenosti s programováním, ale i pro programátory, protože pomáhá vnímat text na více úrovních. Barevně odlišený kód je lépe čitelný a je snazší například zákazníkovi vysvětlit, co jednotlivé části sekvence dělají. Aplikace automaticky zarovnává text, což přispívá k čitelnosti a přehlednosti kódu, jednoznačně ohraničuje bloky kódu a kontroluje syntaxi, čímž přispívá k menší chybovosti při tvorbě skriptů.

Aplikace je vytvořena s myšlenkou znovu-použitelnosti a modularity, a je ji možné rozšířit na jakýkoli automatizovaný měřicí systém. Trendem jsou konfigurovatelná zařízení oproti jednoúčelovým, což přináší velké množství různých druhů konfiguračních souborů, ve kterých je snadné se ztrácet. Rozšíření na další typy konfiguračních souborů by mělo další přínos při konfiguraci těchto systémů.

Literatura

[1] *BUWALDA, Hans, Dennis JANSSEN a Iris PINKSTER. Integrated Test Design and Automation: Using the TestFrame Method. Great Britain: Pearson Education Limited, 2002. ISBN 978-0-201-73725-7.*

[2] MOHANTY, Hrushikesha, J.R. MOHANTY a Arunkumar BALAKRISHNAN. *Trends in Software Testing*. Singapore: Springer, 2017. ISBN 978-981-10-1415-4.

[3] LAMB, Frank. *Industrial automation: hands-on*. New York: McGraw-Hill Education, 2013. ISBN 978-0-07-181645-8.

[4] WANG, Lingfeng a Kay Chen TAN. *Modern industrial automation software design: principles and real-world applications*. Hoboken, N.J.: Wiley-Interscience, c2006. ISBN 978-0-471-68373-5.

[5] WEISFELD, Matt. *The Object-Oriented Thought Process*. Fourth Edition. Upper Saddle River, NJ: Addison-Weasley, 2013. Developer's Library. ISBN 978-0-321-86127-6.

[6] GAMMA, E, R JOHNSON a J VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. ISBN 9780321700698.

[7] FREEMAN, Eric, Elisabeth ROBSON, Kethy SIERRA a Bert BATES. *Head First Design Patterns*. 2nd. United States of America: O'Reilly Media, 2014. ISBN 978-0-5960-07126.

[8] GOYVAERTS, Jan. *Regular Expressions: The Complete Tutorial* [online]. 2007 [cit. 7. 11. 2020n. l.]. Dostupné z: http://www.regular-expressions.info/print.html

[9] VLACH, Jaroslav, Josef HAVLÍČEK, Martin VLACH a Viktorie VLACHOVÁ. *Začínáme s LabVIEW*. Praha: BEN - technická literatura, 2008. ISBN 978-80-7300-245-9.

[10] *LabVIEW™ 2020 Upgrade Notes*. online: National Instruments, March 5, 2020.

Seznam příloh

Příloha v IS EDISON

Adresář elektronických příloh obsahuje položky:

- *launcher.vi* — spouštěcí VI aplikace
- *ukazka.mkv* — video ukázka práce s aplikací
- *project* — adresář s LabVIEW projektem aplikace a všemi konfiguračními daty
- *ATEscripts* — adresář s testovacími skripty automatického testeru